Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) – BarcelonaTech

Degree Final Thesis

From High-Level Languages to Dataflow Circuits

Author: Joaquim Marset Alsina

Director: Jordi Cortadella Fortuny, Computer Science Department

Degree: Bachelor Degree in Informatics Engineering

Specialization: Computer Science

Date: July 2019

Abstract

The traditional way to compute something is writing software that can be executed in the processor's central processing unit (CPU). However, a CPU does not have the computing capacity to properly run applications belonging to certain fields like for example, deep learning and cryptocurrency mining. With the passage of time, graphic processing units (GPUs) began to be used in other fields besides the initially intended ones (e.g. video games), permitting the execution of those applications that CPUs could not. Nevertheless, exists a different way to execute programs or algorithms, that is much more efficient in time and power consumption than executing software in CPUs and GPUs. This other way consists in directly designing and implementing a hardware circuit to particularly execute something, instead of using a general-purpose circuitry that can compute anything.

For this reason, the goal of this project is the development of a synthesis tool that generates data flow circuits from high-level languages. These circuits can be later be implemented in technologies such as field-programmable gate arrays (FPGAs). This project will create a compiler back end, with the help of some existing compiler front end that can translate the initial high-level code into some intermediate representation, such as LLVM. The idea is to have a unique intermediate code for multiple high-level languages. Then, this intermediate representation will be fed to our back end, and it will generate a set of modules with different functions, and channels to transmit data between modules, in the form of directed graphs. Finally, these graphs will be implemented in the mentioned FPGAs, creating the final hardware circuit that will be run. The functioning of these circuits, will follow the data flow paradigm proposed at the MIT in the mid 70's.

Resum

La manera tradicional de computar alguna cosa és creant *software* que es pot executar en la unitat de processament central (CPU) d'un processador. El problema és que una CPU no té la capacitat de còmput suficient per executar correctament aplicacions pertanyents a certs àmbits, com per exemple l'aprenentatge profund o la mineria de cripto-monedes. Amb el pas del temps, les unitats de processament gràfic (GPUs) es van començar a utilitzar en altres camps més enllà dels ideats inicialment (p.e. videojocs), permetent l'execució d'aquelles aplicacions que les CPU no podien. No obstant, existeix una altra manera per executar programes o algorismes, la qual és molt més eficient en el consum de temps i energia que executar *software* en CPUs i GPUs. Aquesta altra manera consisteix a dissenyar i implementar directament un circuit *hardware* per executar alguna cosa en particular, en lloc d'utilitzar un circuit de propòsit general que permet executar qualsevol cosa.

Per aquesta raó, l'objectiu d'aquest projecte és el de desenvolupar una eina de síntesis que generi circuits de *data flow* a partir de llenguatges de programació

d'alt nivell. Aquests circuits es poden implementar en tecnologies com les matrius de portes programables (FPGAs). Aquest projecte crearà el *back end* d'un compilador, amb l'ajuda d'algun *front end* d'un compilador que permeti la traducció de codi d'alt nivell en una representació intermèdia, com per exemple LLVM. La idea és tenir un únic codi intermedi per múltiples llenguatges d'alt nivell. Aleshores, aquesta representació intermèdia es passarà al nostre *back end*, i aquest generarà un conjunt de mòduls amb diferents funcionalitats, i canals per transmetre dades entre mòduls, en la forma d'un graf dirigit. Finalment, aquests grafs s'implementaran en les mencionades FGPAs, creant el circuit *hardware* final que s'executarà. El funcionament d'aquests circuits seguirà el paradigma del *data flow*, proposat en el MIT a mitjans dels anys 70.

Resumen

La forma tradicional de computar alguna cosa es mediante la creación de *software* que se puede ejecutar en la unidad de procesamiento central (CPU) de un procesador. El problema es que una CPU no tiene la capacidad de cómputo suficiente para ejecutar correctamente aplicaciones pertenecientes a ciertos ámbitos como por ejemplo el aprendizaje profundo o la minería de cripto-monedas. Con el paso del tiempo, las unidades de procesamiento gráfico (GPUs) se empezaron a utilizar en otros campos además de los inicialmente ideados (p.ej. videojuegos), permitiendo la ejecución de esas aplicaciones que las CPUs no podían. Sin embargo, existe otra manera para ejecutar programas o algoritmos, la cual es mucho más eficiente en el consumo de tiempo y energía que ejecutar software en CPUs y GPUs. Consiste en diseñar e implementar directamente un circuito *hardware* para ejecutar algo en particular, en lugar de utilizar circuitería de propósito general que permita ejecutar cualquier cosa.

Por esta razón, el objetivo de este proyecto es el de desarrollar una herramienta de síntesis que genere circuitos de *data flow* a partir de lenguajes de programación de alto nivel. Estos circuitos se pueden implementar en tecnologías como las matrices de puertas programables (FPGAs). Este proyecto creará el *back end* de un compilador, con la ayuda del *front end* de algún compilador que permita la traducción de código de alto nivel en una representación intermedia, como por ejemplo LLVM. La idea es tener un solo código intermedio para múltiples lenguajes de alto nivel. De esta forma el código intermedio pasará a nuestro *back end*, y este generará un conjunto de módulos con distintas funcionalidades, y canales para transmitir datos entre módulos, en la forma de un grafo dirigido. Finalmente, estos grafos se implementarán en las mencionadas FPGAs, creando así el circuito *hardware* final a ejecutar. El funcionamiento de estos circuitos seguirá el paradigma del *data flow*, propuesto en el MIT a mediados de los años 70.

Contents

List of Tables

List of Figures

1 Context

1.1. Introduction

This project is a final degree thesis belonging to the specialization of Computer Science, in particular, to the field of compilers. It is a research project, following the investigation of the project's director, Jordi Cortadella.

As a brief description, the goal of this research is demonstrate if combining three existing paradigms, an improvement in both the time and power consumption when executing programs can be achieved. This improvement will be achieved by changing the way compilers generate executable code, and the way this code is later executed. Therefore, we can say that this research trying to change the traditional paradigm of the compilation and execution of programs. However, this degree project will only set the pave to the complete solution that could achieve this change, as some aspects will remain uncovered. Both the full description of the problem and the paradigms the solution is based, will be explained in the following sections.

Formally stating this project, we are going to develop a high-level synthesis (HLS) tool [7] that generates elastic circuits [9][10], which follow the execution model of the data flow architecture [3]. These circuits will be later implemented with technologies such as field-programmable gate arrays (FPGA).

Putting it in a simpler way, we are going to develop some kind of compiler. The front end will be omitted, and we will use an existing framework to verify the initial code and translate it into an intermediate representation. We will center our attention on the back end, that will take this intermediate code, and generate executable code. This executable code will not be the traditional machine code that a central processing unit (CPU) and a graphic processing unit (GPU) understand and execute, but a description of digital circuits in the form of directed graphs. These graphs will be later implemented in the mentioned FPGA and reproduce the behaviour of the initial code, thanks to the fact that the logic blocks of an FPGA can be programmed as the user wants. However, this degree project will end with the generation of those graphs, and leave the programming of the FPGA out of scope.

We will pick whatever program written in some high-level language (HLL) (e.g C, C++ and Haskell), and generate the mentioned directed graphs. The nodes in those graphs will represent the computations the circuit does (e.g. logic and arithmetic operations),

and the arcs will represent communications of data and control signals between the nodes. We will represent these graphs using the language that Graphviz [23] offers, called DOT [18]. Besides, we will use the open-source LLVM compiler framework [22] as the front end of our compiler, that permits the generation of a unique intermediate code (LLVM IR) from multiple HLL.

1.2. Problem's Formulation

We have mentioned above how we are trying to implement the first steps of a solution that could change the traditional way we compile and execute programs. In this section we are going to explain the reasons that have lead to this project.

The last phase of a typical compiler is the generation of the set of instructions that can be finally executed in the processor's CPU, reproducing the behaviour of the initial source code. However, CPUs are not the best solution regarding efficiency and consumption, but their use is too much extended due to their application flexibility and programming ease. We can see in the graphic below the differences between the existing technologies regarding these features.

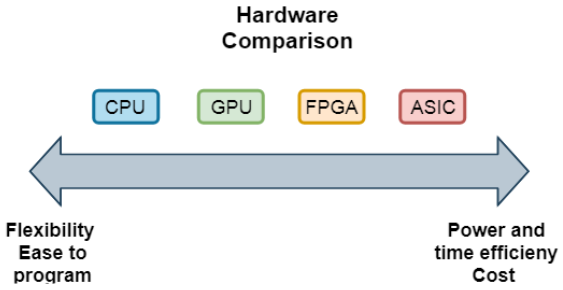

Figure 1: Currently used hardware comparison (Source: own compilation)

We have CPUs in one side and application-specific integrated circuits (ASICs) in the other. ASICs are the most efficient in both time and power consumption, as they are designed for a particular application. But, due to their excessive costs their fabrication implies, their use it is not that extended as with CPUs. CPUs are the cheapest as their purpose is totally general, but they have the mentioned drawbacks. Both GPUs and the FPGAs we intend to use, are in the middle regarding those features, being FPGAs better. FPGAs can be much more expensive than CPUs and GPUs, as well as more difficult to program and less flexible, but their advantages in efficiency and computing capacities, make these drawbacks worth.

Due to this gain in both time and power efficiency, during the last years, some people has been trying to break this current situation and use other kind of technologies to execute certain applications. For example GPUs, initially intended for graphics processing (e.g. video games), have been used for some years to execute applications from other fields like artificial intelligence, machine Learning or deep Learning. Another example could be the more recent use of ASICs in cryptocurreyncy mining. Using such technologies have permitted the acceleration of critical parts of some applications and algorithms, compared to their execution in a CPU, or even completely execute some of them that in a CPU was unfeasible.

Therefore, the project we are going to develop follows the same idea of trying to develop a solution to break this traditional situation, and accelerate the execution of programs and reduce its power consumption. In our case we want to achieve this result by introducing the novelty of combining some paradigms that others alternatives have not considered, and using technologies such as FPGAs to implement the resulting hardware circuits.

1.3. Stakeholders

In this section we are going to explain the different persons that will be involved in the project development, as well as those that might be involved in the future, once we finish our work, that can benefit with the work we have done. Also, we will consider those who can benefit once the complete solution is finished, beyond this degree thesis scope.

1.3.1. Developer

The developer will be this final degree thesis' author. He will be the one who will perform all the different tasks to develop the entire project. These tasks do not only include the development of the mentioned compiler back-end, but they also include the writing of the needed documentation, preparation, and exposition of the project's defense, planning the complete development, creation of a precise budget, analysis of sustainability, and constant communication with the project's director.

1.3.2. Director of the project

The director of the project will be Jordi Cortadella, professor of the Computer Science Department. He is the one who proposed the topic for this project, based on a research he was taking part. He will act as the client, giving the requirements that the solution must satisfy, as well as the one who will help the developer, giving advice, and evaluating his work. A constant communication will be hold with him, via meetings or emails, and he will be up to day during the whole development.

1.3.3. Users

In this set we include those who can benefit from the development of this project. As we are trying to offer an alternative to solve a problem, that others have not tried, we cannot ensure its success. Besides, we will not develop the complete solution, and only perform the first steps. For these reasons, the ones who can benefit are researchers who might be interested in the approach we present, and want to complete the solution or use it as base for its own approach. For example, the director of the project can also be included, as he is the most interested in developing this approach, and he can continue with this project and finish it.

If the complete solution could achieve positive conclusions that could lead to a useful approach to solve the problem, we could also include software and hardware developers, as we could make their work easier. We could help software engineers who are not used to hardware primitives, accelerating their software. And we could help hardware engineers, facilitating the design and verification of efficient hardware, as we could offer a higher level of abstraction of the circuits' design. Besides, we could also include common people, who would benefit in an indirect and transparent way. For this to happen it would mean the addition of a more efficient hardware, like the one we have mentioned before, in common computer's processors. But imagining this situation happening is quite difficult at this moment.

1.4. State of the Art

As explained in the contextualization, this is a research project that tries to provide an approach to solve the problem, combining three paradigms. These three paradigms are the data flow architecture, the high-level synthesis, and the elastic circuits.

1.4.1. Data Flow Architecture

Proposed by Jack Dennis from the MIT, as an alternative to the classical von Neumann computer architecture. The main difference is that the data flow architecture does not have a program counter sequencing the execution of the different instructions. They are executed as soon as the input arguments are available, and the output is free of previous data. This permits the execution of instructions out of order, and the achievement of high levels of parallelism when executing programs.

This architecture was designed to utilize the data flow language, Dennis previously proposed, as its base language [2]. Starting from a program written in some programming language, this was translated into this data flow language, and later executed in the machine. A program in this language is a directed graph where the nodes represent different operators (e.g logic and arithmetic operations), and the arcs represent transmission of data and control signals. This language is quite simple, but it permits the execution of conditionals and iterations. It was based on the work of R. M. Karp and R. E. Miller [6], and the work of J. E. Rodriguez [5]. Both of them presented models for parallel computations consisting in directed graphs. Rodriguez in fact is considered as the one who created the data flow graphs, with his model called Program Graphs. Below we have an example of a program with its corresponding data flow graph.

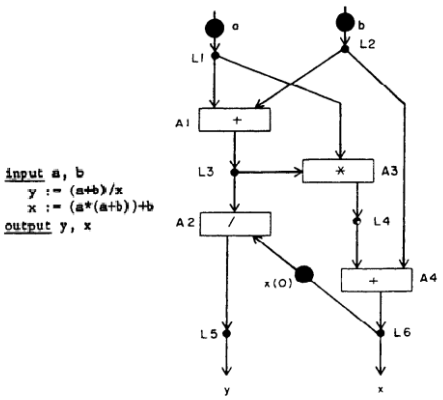

Figure 2: Example program with its corresponding data flow graph (Source: [3])

In this architecture, a program's data flow graph is held as a collection of instruction cells, where each cell contains four registers with the operation code to be executed, the input arguments of that operation and the addresses of the output destination registers. All these cells are stored in the memory. When a cell contains the needed arguments, and the output is empty, the operation is ready to be executed. The cell contents are transmitted as an operation packet to be delivered to a functional unit Dennis called

Operation Unit that will execute the corresponding instruction. Then, data packets with the result are generated, and delivered to the corresponding registers, enabling new instructions to be executed. As there is no program counter, and many instructions can be activated simultaneously, a good management is mandatory. For this reason the architecture has two more functional units, that Dennis called them Arbitration Network, some kind of fetch unit, and Distribution Network, some kind of update unit. The first one in charge of delivering the operation packets to the corresponding Operation Unit, and the second one in charge of delivering the result packets to the recipient registers [3][4].

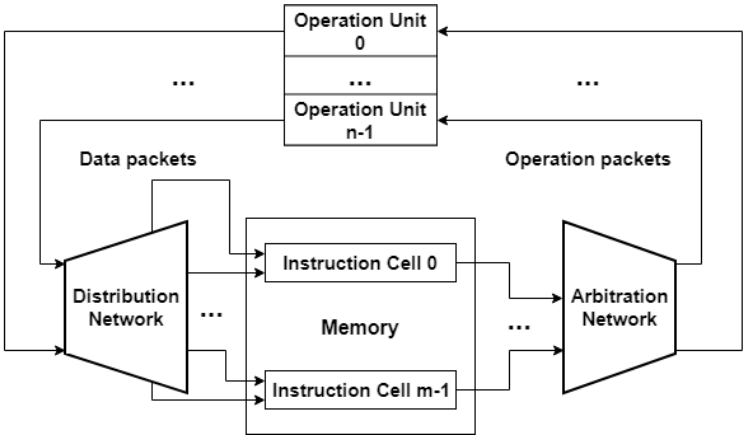

Figure 3: Data flow architecture model (Source: [3])

This architecture was proposed in the 70s, but it was unfeasible and finally left aside due to the development and cost of the hardware at that time. However, with the new age of the multi-core processors, this architecture model is studied and investigated again as it has overcome the past hardware problems, and it has become more feasible.

1.4.2. High-Level Synthesis

High-level synthesis (HLS) [7] consists in translating a high-level language source program, into a description of a digital circuit that implements the same behaviour. This description is done using some hardware description language (HDL), which looks like a programming language. However, the description of the circuit is not done at the level of logic gates, but at a higher level of abstraction known as Register-Transfer Level (RTL), that is more similar to high-level programming.

HLS tools can eliminate the need of the hardware designers to translate manually from a high-level specification into a RTL description, and obtain better results than manual HDL coding. The use of HLS has increased during the last years, due to the growing

complexity of embedded systems, reducing the work of the hardware designers and the later verification of these resulting circuit designs. Besides, HLS makes the circuit design accessible for software developers too, who are not much familiar with the hardware primitives, and who outnumber hardware designers.

HLS involves three main steps, i.e. allocation, scheduling, and binding.

- The allocation phase decides the number of the different functional units, storage components, and connectivity components that will be needed to correctly perform the different computations of the initial source code while obeying the design constraints.
- The scheduling phase determines which operation executes at each clock cycle, based on the number of components of each kind, the clock cycle period and the possible data dependencies between these operations.
- The binding phase assigns all the components decided at the first phase to the different computations scheduled in the second phase, in a non-overlapping and optimized way. That is, assigning the different variables to the storage units, the different operators to the functional units, and the different transmissions between components to the connectivity units.

All those components are selected from what is called RTL component library, that contains the different specifications of each component, like the delay of a component. It is an input for the HLS, together with the clock frequency of the target machine. Both of them, plus other design constraints, like latency, throughput, and area, will determine the resulting RTL description.

Onces the three main steps are performed, it comes the generation of the RTL model, implemented as a data path and a controller. A data path consists of a set of storage elements (e.g. registers and memories), a set of functional units (e.g. ALUs, multipliers and shifters), and communication elements (e.g. buses and multiplexers). The controller is a finite state machine that controls the flow of data in the data path by setting the value of the control signals in the data path components like a multiplexer.

Currently there are many HLS tools out, and for example LegUp [8] (`http://legup.eecg.utoronto.ca/`) and Vivado HLS tools (`https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`) are two important ones. Both of them can synthesize C code into FPGA circuits, accelerating its execution. Also, both of them use the same LLVM framework we intend to use.

1.4.3. Elastic Circuits

The elastic circuits [9][10], proposed among others by this project's director, consist in digital circuits with a property known as *elasticity*. This property refers to the ability of a circuit to adapt its activity to the different timing delays of the computations and communications it does.

This paradigm appears due to the way synchronous circuits, governed by a clock, deal with these different delays. They deal with these delays by forcing a big clock period as a security margin to ensure that in one clock cycle whatever computation or communication can be finished. Asynchronous circuits can deal better with this problem, but due to the lack of a good design flow and tools that are essential in the design of synchronous circuits (i.e. electronic design automation (EDA)), asynchronous circuits are still behind synchronous circuits. However, elastic circuits can be properly designed with the current EDA flow design.

Elastic circuits wait the necessary delays for the different inputs to arrive, and then the operation is performed. Therefore, we have to consider the accumulated delay of the arguments' arrival plus the operation delay. But, this accumulated time is not fixed and it changes depending on the operation, so they avoid losing some time like synchronous circuits do with their security margin. Elastic circuits can use this model of executing instructions, thanks to a synchronization protocol they have. Either they are synchronous and the synchronization it is done by the clock, permitting idle clock cycles of waiting, or asynchronous and then some handshake signals are needed.
Besides, the communication of data between components depends on two signals with different directions. One is transmitted from sender to receiver indicating that the sender is transmitting valid data. The other is transmitted the other way round indicating that the receiver can consume the data. This communication of control signals to determine the transmission of data is what permits the latency-insensitivity that characterizes the elastic circuits.

1.4.4. Combination of them

In the high-level synthesis, we have seen how the different operations are scheduled statically during the synthesis. This forces the tool to make a conservative plan that will avoid problems with control and data dependencies, but it will also suppose a slow down in performance. Instead, if we decided to dynamically schedule the operations as soon as they are ready, like we have seen in the data flow paradigm, we could achieve a better parallelism. And the mentioned elastic circuits are a good choice, as they behave in this desired way of executing operations. We only need some way to represent and describe these circuits by software. The data flow graphs that we have also presented are a good option, as there is a literal translation between the components of those directed graphs to the different components of the hardware circuits, only extending these graphs by adding some control components used in the elastic circuits.

This is the reason why we are combining them to produce a possible solution that could solve the problem.

These three paradigms have existed for a few years, and different works have been proposed. Below we mention some of them that could be considered similar to this project to a greater or lesser degree.

Some of them share the same goal of trying to accelerate critical parts of programs, synthesizing programs into hardware circuits, like LegUp [8] and Vivado HLS tools, mentioned before. Also, the work mentioned in the paper cited in [14] has the same goals and their approach is quite similar, but it uses another intermediate language, and it generates asynchronous circuits instead of elastic.

There are others, that combine both the high-level synthesis and the elastic circuits, but with the purpose of easing the burden to the hardware engineers when designing circuits with an ever-increasing complexity. For example the paper cited in [12], and the PhD thesis cited in [13].

Finally, we want to mention the *Ecole Polytechnique Fédérale de Lausanne* (EPFL), with whom the director of the project was collaborating. They are the closest to this project and the biggest reference, as they used the same paradigms and approach, and tried to solve the same problem [1].

2 Programming Tools

In this section we are going to explain the two main tools needed to program the solution, i.e., the LLVM compiler framework and the DOT language.

2.1. LLVM Compiler Framework

LLVM [22] is an infrastructure to develop compilers. It started as a research project to provide a SSA-compilation strategy to support dynamic and static compilation of arbitrary programming languages. Since then, it has grown a lot, and it has become a big open-source project, containing sub-projects that are being used in the commercial and academic field, as everyone can use it to develop their own tools. And thanks to the fact that everyone can use it without many limitations, today exists multiple front ends that permit the compilation of multiple high-level languages such as C, C++, Python, Haskell, Java or Julia. We have decided to use it precisely because it can compile lots of high-level languages into a unique intermediate representation, making the steps of the development totally independent of the initial language.

This intermediate representation is called LLVM IR. It is a low-level programming language similar to assembly, strongly typed, with a reduced set of instructions, with an infinite number of registers, usually in Static Single Assignment (SSA) form. SSA is a property of intermediate representations that requires that each variable is assigned only once. Therefore, every time a variable is assigned in the original code, in the SSA form supposes a new variable, or what we could call a version of that variable. In order to work with the SSA form we need the ϕ functions. A ϕ function is an statement used to resolve the version of a variable, in some execution point, when it could have come from several independent execution paths. The SSA form is a good property the intermediate languages have, as it facilitates and improves the analysis and optimization that the compiler's back end performs.

Another feature LLVM has is the possibility of applying optimizations and analysis to the generated intermediate representation. They are run with the LLVM tool that is called with the *opt* command, and in the form of what LLVM calls Passes [15][16]. Passes traverse the LLVM IR gathering information, transforming the code, or performing some function that cannot be considered as analysis nor transformation. Passes can be applied at different levels, like the entire program, each function, each basic block or each loop. Besides, different passes can be chained, and we can create interactions between them, forcing a sequential execution, as well as prohibit executing some of them together.

We have decided to program our solution using LLVM passes. With passes we can easily traverse the different functions inside the LLVM IR. Also, as we can chain passes, we can directly get the information computed in analysis passes, without the need of storing in some file and reading it again. Another good point is that executing passes is very easy, and only two or three command line will be needed to execute our solution, using the mentioned tool. We will use some existing passes, as well as creating some of them, to perform some transformations and analysis to the generated LLVM IR. It is mandatory that the passes are executed in the order we present them, in order to avoid conflicts between those making transformations, and ensure the correct results on those gathering information. Below we explain the functionality of each of them.

2.1.1. mem2reg

Transform pass that promotes memory references to register references. Some LLVM front ends do not generate LLVM IR in SSA form. The alternative to the SSA form, that avoids the ϕ functions and having multiple versions of the same variables, is to store the variable in the stack, instead of using registers. However, this suppose having each local variable stored in memory, and every time its value is read or written, this suppose memory traffic with loads and stores. Hence, the simplest and common operations, like an addition, supposes some memory traffic and a performance problem. Therefore, this pass transform a non-SSA LLVM IR that uses stack variables, into a SSA LLVM IR. Within the elastic circuits, we follow the approach of the SSA as it is more efficient, and it is more natural, as each time we assign a variable its

value it is hold by a different module, so we need the ϕ functions to receive the data from the correct one. Below we present an example.

```
define dso_local i32 @testMem2Reg(i32 %a, i32 %b)
    #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %cmp = icmp sgt i32 %0, 10
  br i1 %cmp, label %if.then, label %if.else
if.then: ; preds = %entry
  %1 = load i32, i32* %b.addr, align 4
  %mul = mul nsw i32 %1, 2
  store i32 %mul, i32* %b.addr, align 4
  br label %if.end
if.else: ; preds = %entry
  %2 = load i32, i32* %b.addr, align 4
  %mul1 = mul nsw i32 %2, 4
  store i32 %mul1, i32* %b.addr, align 4
  br label %if.end
if.end: ; preds = %if.else, %if.then
  %3 = load i32, i32* %b.addr, align 4
  ret i32 %3
}
```

```
define dso_local i32 @testMem2Reg(i32 %a, i32 %b)
    #0 {
entry:
  %cmp = icmp sgt i32 %a, 10
  br i1 %cmp, label %if.then, label %if.else
if.then: ; preds = %entry
  %mul = mul nsw i32 %b, 2
  br label %if.end
if.else: ; preds = %entry
  %mul1 = mul nsw i32 %b, 4
  br label %if.end
if.end: ; preds = %if.else, %if.then
  %b.addr.0 = phi i32 [ %mul, %if.then ],
    [ %mul1, %if.else ]
  ret i32 %b.addr.0
}
```

Figure 4: Example of *mem2reg* pass (Source: own compilation)

The original source code was simply a function receiving two parameters, and depending on the value of the first parameter, the second was multiplied by a different constant, and finally returned. In the left code, we have the version without applying the pass. The *alloca* instruction is in charge of reserving space in the stack, returning a pointer to that memory address. We can see how it has to store in memory copies of the parameters, in order to use them, and every time they are used or assigned, a load or store is required. However, the right code, where the pass has been applied is much shorter and more natural. We can see how the modified value of the second argument, in each branch of the if-else, is stored in a temporal, and outside the if-else, a ϕ is required to receive the value of the path executed in order to return it. Also, the temporal storing the result of the ϕ, is a new one, due to the SSA principle of only one assignment.

2.1.2. lowerswitch

Transform pass that lowers a switch instruction into the corresponding set of conditions and branches. In the elastic circuits we cannot directly implement a switch instruction with the modules we have, so we have to reduce it. We present an example below.

```
define dso_local i32 @testSwitch(i32 %a, i32 %b)
    #0 {
entry:
  switch i32 %a, label %sw.default [
    i32 1, label %sw.bb
    i32 2, label %sw.bb1
  ]
sw.bb: ; preds = %entry
  %mul = mul nsw i32 %b, 2
  br label %sw.epilog
sw.bb1: ; preds = %entry
  %mul2 = mul nsw i32 %b, 4
  br label %sw.epilog
sw.default: ; preds = %entry
  br label %sw.epilog
sw.epilog: ; preds = %sw.default, %sw.bb1, %sw.bb
  %x.0 = phi i32 [ 2, %sw.default ],
  [ %mul2, %sw.bb1 ], [ %mul, %sw.bb ]
  ret i32 %x.0
}
```

```
define dso_local i32 @testSwitch(i32 %a, i32 %b)
  #0 {
entry:
  br label %NodeBlock
NodeBlock: ; preds = %entry
  %Pivot = icmp slt i32 %a, 2
  br i1 %Pivot, label %LeafBlock, label %LeafBlock1
LeafBlock1: ; preds = %NodeBlock
  %SwitchLeaf2 = icmp eq i32 %a, 2
  br i1 %SwitchLeaf2, label %sw.bb1, label %NewDefault
LeafBlock: ; preds = %NodeBlock
  %SwitchLeaf = icmp eq i32 %a, 1
  br i1 %SwitchLeaf, label %sw.bb, label %NewDefault
sw.bb: ; preds = %LeafBlock
  %mul = mul nsw i32 %b, 2
  br label %sw.epilog
sw.bb1: ; preds = %LeafBlock1
  %mul2 = mul nsw i32 %b, 4
  br label %sw.epilog
NewDefault: ; preds = %LeafBlock1, %LeafBlock
  br label %sw.default
sw.default: ; preds = %NewDefault
  br label %sw.epilog
sw.epilog: ; preds = %sw.default, %sw.bb1, %sw.bb
  %x.0 = phi i32 [ 2, %sw.default ],
    [ %mul2, %sw.bb1 ], [ %mul, %sw.bb ]
  ret i32 %x.0
}
```

Figure 5: Example of *lowerswitch* pass (Source: own compilation)

The original code of this example was a function receiving two parameters, and depending on the value of the first one, the result of the function was modified, using the second parameter, and a constant changing depending on the switch case. We want to mention that in the left code we previously applied the *mem2reg* pass. In the left code we can see the switch instruction, being the first label the code to execute in the default case, and then the different cases depending on the value of the first parameter. The right code is quite simple, comparing the value of the tested argument against the possible values of the different cases, jumping to the corresponding section if the value is found, or to the default case if none of them are the correct value.

2.1.3. gepLowerPass

Transform pass that lowers the instruction *getelementptr*, used to compute the memory address of array elements or structure fields, into a set of arithmetic and cast instructions. This instruction accepts a pointer to the structure as one parameter, and a set of indices to move inside the structure. The first step the pass does is transforming the input pointer into an integer, in order to properly make additions, multiplications and left shifts. Then, for each index, we compute the offset this index represents, with respect to the initial address of type that this particular index

tries to index, and we add it to the address calculated so far. We want to mention that all the constant offsets are accumulated, and added in the end. This way we compress all the additions of constant values into only one LLVM IR instruction, meaning less elastic modules. Once the final address is calculated, the integer is transformed again into a pointer of the type the final address is pointing to, resulting in the output pointer that the original instruction obtains. Below we present an example.

```
define dso_local i32 @testGEP([32 x i32]* %a) #0 {
entry:
  %arrayidx1 = getelementptr inbounds [32 x i32],
    [32 x i32]* %a, i64 10, i64 23
  %0 = load i32, i32* %arrayidx1, align 4
  %mul = shl nsw i32 %0, 1
  ret i32 %mul
}
```

```
define dso_local i32 @testGEP([32 x i32]* %a) #0 {
entry:
  %0 = ptrtoint [32 x i32]* %a to i64
  %1 = add i64 %0, 1372
  %3 = inttoptr i64 %2 to i32*
  %4 = load i32, i32* %3, align 4
  %mul = shl nsw i32 %4, 1
  ret i32 %mul
}
```

Figure 6: Example of *gepLowerPass* pass (Source: own compilation)

The original code was a function receiving a matrix as its argument, and returning the value of a particular element multiplied by a constant. In the left code we have first applied *mem2reg* and *lowerswitch*, but the second pass has not effect in the original LLVM IR code. We can see the *getelementptr*, receiving a pointer to the matrix, and two indices indicating in this case the row and column it tries to access. In the right code we have lowered this instruction with the method explained above.

2.1.4. instnamer

Utility pass that gives names to all the anonymous temporaries used to store results of computations. Instructions in the LLVM IR are represented by a class called Instruction, subclass of another one called Value, that represent constants, instructions, arguments, and labels. Each instance of the Value class have a name. In the case of the instances of the Instruction class, the name belongs to the possible temporary used to store the result of the instruction. And temporaries of the form %0, %1, etc, do not have a name. They are not essential, but the names are used for printing purposes, like for example printing the results of the live variable analysis, explained in the following section. Below we show an example, where the only change is the name in the mentioned temporaries.

```
define dso_local i32 @testGEP([32 x i32]* %a) #0 {
entry:
  %0 = ptrtoint [32 x i32]* %a to i64
  %1 = add i64 %0, 1372
  %2 = inttoptr i64 %1 to i32*
  %3 = load i32, i32* %2, align 4
  %mul = shl nsw i32 %3, 1
  ret i32 %mul
}
```

```
define dso_local i32 @testGEP([32 x i32]* %a) #0 {
entry:
  %tmp = ptrtoint [32 x i32]* %a to i64
  %tmp1 = add i64 %tmp, 1372
  %tmp2 = inttoptr i64 %tmp1 to i32*
  %tmp3 = load i32, i32* %tmp2, align 4
  %mul = shl nsw i32 %tmp3, 1
  ret i32 %mul
}
```

Figure 7: Example of *instnamer* pass (Source: own compilation)

2.1.5. liveVarsPass

Analysis pass that computes the live variables analysis [11] of the set of basic blocks [11] inside each function. This analysis is typically computed in the optimization phase of the compiler, and it is used to apply some optimization like dead-code elimination. The information of this passed will be used to generate a specific type of elastic module, that will be necessary to properly generate the data flow graphs. The result of this analysis will be directly used by the following pass, as well as printing them in a text file. All the concepts involved with this analysis will be later explained.

2.1.6. dfGraphPass

Utility pass that generates a data flow graph, describing an elastic circuit, from the LLVM IR fed as input. This is the pass that traverse each instruction inside each function and generates the needed elastic modules (i.e. nodes) and elastic channels (i.e. edges). The graph will be described using the graph description language called DOT, that will be explained in below. The result of this pass will be a file containing the description of the resulting graph.

2.2. Graphviz

Graphviz [23] is an open source software to create and visualize graphs. It offers a way of representing structural information as graphs and networks, and has important applications in fields like networking, bioinfomartics, software development, database and web design, or machine learning. The graphs are created with the help of the DOT language, a graph description language that permits the creation directed and undirected graphs.

The DOT language [18] has a very simple syntax, that only needs to create the graph and fill it with nodes and edges. Moreover, it offers the possibility to add custom attributes to the graph, nodes and edges. Below we present a dummy graph created with the DOT language.

```
digraph testGraph {
    label = "Test Graph";
    Node1[color = blue, attr1 = 1, attr2 = "abc"];
    Node2[style = filled, fillcolor = red];
    Node3[shape = box];
    subgraph cluster_SG {
        label = "Subgraph"
        Node4;
        Node5;
        Node6;
        subgraph cluster_SG_ {
            label = "Subgraph's subraph"
            Node7;
            Node8 [shape = diamond];
        }
    }
    Node1 -> Node2;
    Node2 -> Node3;
    Node4 -> Node7 [color = magenta];
    Node2 -> Node8 [style = dashed];
    Node6 -> Node1 [penwidth = 5];
    Node4 -> Node5;
}
```

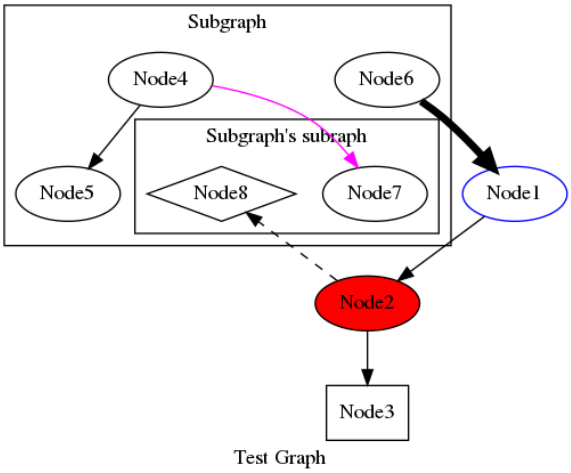

Figure 8: Example graph created with the DOT language (Source: own compilation)

As we can see, the graph is created specifying if it is a digraph or a common graph, and then specifying its list of nodes and edges. Inside a graph, the nodes and edges can be grouped in sub-graphs, using the corresponding sub-graph directive. As we can see, the nodes in a sub-graph are place inside a black frame in the resulting graph. Also, we can observe how the attributes are specified inside square brackets, with the typical pair of name and value. The edges can be specified with an arrow for a directed edge, or two hyphens for an undirected edge.

However, this language is quite simple as its purpose is offering a way of describing graphs, but it does not provide any type of infrastructure to render and manipulate those graphs. For example, the attributes added do not have any effect by themselves, but they can be used by other applications like Graphviz. These applications offer tools to process the DOT files, permitting its rendering and manipulation. Besides, Graphviz offers the possibility to customize the graph with various attributes that can be added to the graph, nodes or edges, that will affect the rendered graph. These include for example attributes to modify the aesthetics of the graph, like the ones we have added in the different nodes. Also, they offer the possibility of exporting the graphs in useful formats like PDF or Postscript.

In this project we are going to use the DOT language to represent the data flow graphs we generate, as they can be easily defined and represented, as well as easily visualized. The idea is to generate a graph for each program, and create sub-graphs for each function and each basic block inside each function. We are going to define some particular nodes, where all of them will share some common attributes, as well as certain types will have some exclusive attributes.

Besides, we will try to take profit of the customization attributes Graphviz offers, to facilitate the visualization and understanding of the resulting graphs.

3 Live Variables Analysis

In this section we are going to explain the concept and algorithm of the live variable analysis, learned during the Compilers course from the Computer Science Specialization. This analysis is done in the LLVM pass mentioned before, called *liveVarsPass*. However, we will first explain another concept that is related with.

3.1. Control-Flow Graph

The control flow represents the order in which the different instructions, statements, or function calls, are executed in an imperative language. Within this type of languages we have instructions that can break the execution flow , permitting the program to start executing a new path depending on a condition.

Therefore, a control-flow graph is a directed graph that describes the different execution paths a program can take. In this graph a node represents a basic block, and an edge represents the existence of a control flow statement in the origin node, that changes the execution flow, and starts executing the instructions in the destiny node. A basic block [11] is a set of instructions that are executed sequentially, and no control flow statements can be found except at the last instruction. Therefore, the control flow can only enter the basic block through the first instruction in the block, and it can only exit the block through the last instruction.

In one of its intermediate phases, a compiler generates an intermediate code from the input code, that has a lower level of abstraction. In this code, the compiler applies optimizations before translating it to the executable machine code. Typically, this intermediate code is divided into the set of basic blocks in order to apply these optimization. The process used to identify the set of basic blocks consists in traversing the code, and identify the leaders of each basic block. They delimit the different basic blocks, and they are always the first instruction of a basic block. The instructions that can act as leaders of basic blocks are the first instruction of the code, the destination instruction of a jump, and the instruction following a jump. Once the leaders are identified, the only thing missing is group together the instructions from one leader until the one before the next leader. The last instruction of the basic block is always a control flow statement, and the more typical ones are conditional/unconditional branches and returns.

The intermediate code LLVM generates (LLVM IR), that we intend to use in our back-end, is divided in basic blocks. Therefore, all the work done will be at the level of basic blocks, including the generated data flow graphs that will be divided into basic blocks.

We are going to show an example of a control-flow graph using the code of a function that calculates the factorial of a positive integer passed as argument. The code has already transformed into the LLVM IR, and also transformed with the mentioned passes. We will use the same example to show the process of generating the data flow graph.

```
define dso_local i32 @fact(i32 %n) #0 {
entry:
  br label %while.cond
while.cond: ; preds = %while.body, %entry
  %fact.0 = phi i32 [ 1, %entry ],
    [ %mul, %while.body ]
  %i.0 = phi i32 [ 2, %entry ],
    [ %add, %while.body ]
  %cmp = icmp sle i32 %i.0, %n
  br i1 %cmp, label %while.body,
    label %while.end
while.body: ; preds = %while.cond
  %mul = mul nsw i32 %fact.0, %i.0
  %add = add nsw i32 %i.0, 1
  br label %while.cond
while.end: ; preds = %while.cond
  ret i32 %fact.0
}
```

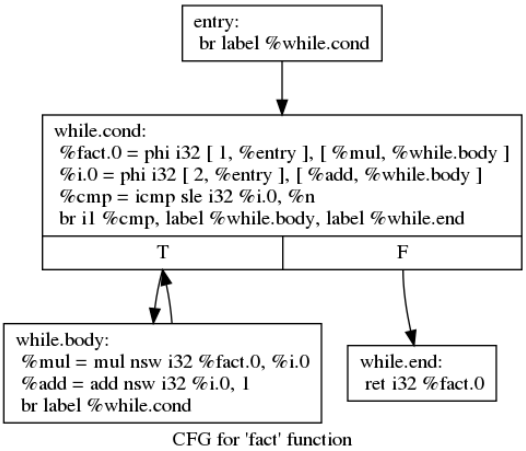

Figure 9: Example of a control-flow graph (Source: own compilation)

3.2. Live Variables Analysis

The live variable analysis [11] is a type of data-flow analysis, performed during the optimization phase of a compiler, as it is needed to perform certain optimizations. These analysis derive information about the values of the different variables that appear in the program, in different points of the program. These program points, are locations within the program, before or after executing a particular instruction. However, most of these analysis are performed at the level of basic blocks, gathering, for each basic block, information before executing the first instruction in the basic block, and after executing the last instruction of the basic block.

In the data-flow analysis we have $In[s]$, that represent the value of the different variables before executing the statement s, and $Out[s]$ that represent the values after executing s. Exists a relationship between $In[s]$ and $Out[s]$, called transfer function, where one can be expressed in terms of the other. If the analysis is performed forward, $Out[s]$ depends on $In[s]$, and the other way round if the analysis is performed backwards.

Performing these analysis means solving for each basic block, two equations, that change if the analysis is done forward or backward.
Forward analysis:
$Out[B] = f_B(In[B])$

$In[B] = \bigcup_{P \text{ predecessor of } B} Out[P]$

Backward analysis:

$In[B] = f_B(Out[B])$

$Out[B] = \bigcup_{S \text{ successor of } B} In[S]$

The f represents the transfer function, that changes for each data-flow analysis.

The idea is to perform an iterative algorithm, solving these equations for each basic block until there is no change in the result of any basic block.

The live variable analysis is a backward data-flow analysis, where the information computed for each basic block represents the set of variables that are *live* at the beginning and at the end of the block. A variable is alive at a program point, if it holds a value that might be used in some other program point, reachable from the first program point, before it is written again.

In order to perform this analysis, we need to compute, for each basic block, the set of variables that are used in the block before any assignment in the same block, and the set of variables that are assigned before any use. These two sets will be used in the transfer function. As it is a backward analysis, it starts from the last basic blocks of a function, assuming that the set of live variables at the end of the last basic block is empty.

The data-flow equations that we must solve in this analysis are the following:

$In[B] = Use_B \cup (Out[B] - Def_B)$

$Out[B] = \bigcup_{S \text{ successor of } B} In[S]$

And as we have said, we have to iterate these equations, starting with these two sets being empty for each basic block, until there is no change in the content of any set.

In this project we are going to use the live variable analysis to generate the data flow graphs. The idea is to place certain components in each basic block to properly receive and send data from one basic block to another, in specific situations.

- When a block ends with a conditional branch, and a variable is alive at the end of that basic block, we have to ensure that the successor that will execute, receives the variable.
- When a basic block uses a variable that comes from multiple predecessors, and these predecessors are totally independent and only one can execute, we must ensure that the basic block receives the variable from the executed predecessor.

If we were not to control these situations, we would end with an operation waiting eternally for an operand to come, and never being able to execute (i.e. a deadlock).

However, we will apply a slightly modification to the algorithm that computes the analysis, that will facilitate its results and the latter generation of the data flow graph graph. Following the live variables analysis in a phi function, the different versions of a variable reaching that basic block can be considered uses inside the phi function. Therefore, as these instructions are placed at the beginning of the block, all these variables are alive at the beginning of the block. Therefore,

we will add them to the set of live variables at the end of each predecessor. As each of these variables comes from a particular block, those that do not come from that predecessor will be considered alive at whatever point inside that block, as they will not be written there. Therefore, we are introducing useless variables, that do not have any importance in some blocks. For this reason, we have decided to only add each different version of the variable being processed with a phi function, to the set of live variables at the end of the only predecessor that defines it, indicated in the same phi function.

Following with the example presented in the control-flow graphs, here we present the live variable analysis of the function present in that example, where the mentioned modification in the algorithm that computes the live variable analysis has been applied.

```
Block entry
    Live In
        n
    Live Out
        n
Block while.cond
    Live In
        n
    Live Out
        n
        fact.0
        i.0
```

```
Block while.body
    Live In
        n
        fact.0
        i.0
    Live Out
        n
        mul
        add
Block while.end
    Live In
        fact.0
    Live Out
```

Figure 10: Results of a live variable analysis (Source: own compilation)

4 Data Flow Graph Components

In this section we are going to describe the different nodes and edges that will form the data flow graphs we generate. As a remainder, these graphs describe elastic circuits that can be later implemented in an FPGA. The nodes in the data flow graphs represent the different modules in the elastic circuits that perform different functions. These modules are synchronous and equivalent to data path components (e.g. Arithmetic Logic Unit (ALU) and Floating-Point Unit (FPU)). However, they implement latency-insensitivity by communicating with their predecessors and successors through a pair of handshake signals, usually referred as *valid* and *ready*, instead of plane wires. The first one goes from sender to receiver to indicate the sending of a valid piece of data. The second one goes from receiver to sender to indicate the availability of receiving a new piece of data. As we have explained in the state of the art, these handshake signals are used to achieve the property of *elasticity*, and have a circuit that can tolerate the variations in delays in both computations and communications. Achieving this property avoids the design problems of asynchronous circuits, or the wide clock periods in synchronous circuits.

Elastic modules have some common attributes that include the delay to execute its function, as

well as a list of input and output ports that receive and transmit data. As we explained before, once a module receives all the needed data through its input ports, and the output ports are all free, the module can be executed. Each port has some features, that include the width of data they accept and the delay of the data to go through that port. Different types of modules will have different number of input and output ports. Besides, some of these modules will have specific attributes that will define its behaviour.

The transmission of data between the output port of a module and an input port of another module, are called elastic channels, and they represent the edges in the data flow graphs. Each transmission communicates the handshake signals, but data can also be transmitted. For this reason, we can distinguish three types of data communication: transmission of boolean data (i.e. 1-bit data), other type of data (i.e. > 1-bit data), or only the control handshake signals (i.e. 0-bit data). As we will have this three mentioned type of channels, we will distinguish them using different colors with the attributes available in the DOT language.

In order to define all these modules, we have used the concept of inheritance and polymorhism, defining them as a hierarchy of classes and sub-classes. We have considered it as the easiest way to deal with modules connecting with others, without they types of each of them, as well as the best way to represent the common attributes all the modules share. Besides, in order to create the DOT graph, we have defined common and particular methods to print the nodes and channels, using the DOT syntax.

In the following subsections we explain all the elastic modules. We want to mention that in all the figures showing the module, a red arrow represents a channel transmitting only the handshake control signals, the pink arrow represents also the transmission of only boolean data, and the blue arrow represents transmission of whatever type of data and even control signals. Some modules are used to transmit data, but also control signals.

4.1. Operator

In the initial specification, they were used to implement arithmetic, logic, and relational operations. However, we have decided to use them to implement multiple types of instructions in LLVM IR. All the conversion instructions that are used to modify the type of a value, the memory access and addressing instructions, and the vector operations instructions, are represented with this module. This means that we have to indicate the operation that it is performing. And this also means that this module can have one, two or more input ports, and a maximum of one output port, depending on the instruction. However, most of the instructions have one or two input ports and one output port.

Operator modules can be pipelined, meaning that they can start executing another operation before ending the previous one. The behaviour of a pipelined operator is determined by two parameters, i.e. latency and initiation interval, both measured in number of cycles. Latency

measures the number of cycles to perform an operation, whereas initiation interval measures the number of cycles before another operation can start. When latency is 0, the block is assumed *combinational*, and it cannot be pipelined. When latency is greater than 0, the block is *sequential*, it can be pipelined, and the value of the initiation interval has the mentioned behaviour. In pipelined units the delay of the ports and the block have a different interpretation. The delay of the input ports represent the delay from the port to the internal registers of the block, the delay of the output ports represent the delay from those internal registers to the output ports, and the delay of the block represent the delay from register to register.

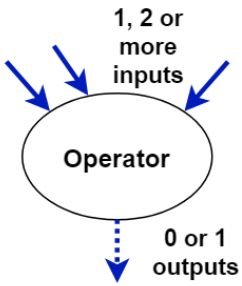

Figure 11: Operator module (Source: own compilation)

4.2. Buffer

A buffer is used to store data, implemented as a First in, First out (FIFO) register, with a certain capacity. This module only contains one input port and one output port, that can transmit whatever type of data.

Buffers are characterized by two parameters, i.e., size and transparent. The size is simply the capacity of the buffer, whereas the second one indicates if the buffer is transparent and it can be by-passed. Buffers are placed on the edges of the data flow graph, and a transparent buffer will only store data from the source node if the destiny node cannot accept it. A non-transparent buffer will receive all the data from the source node of the edge, and it will store it until it can send it to the destiny node in later clock cycles.
If some data is transmitted, but the destiny node cannot it, the output port of the first node will not get empty, and further executions of this node will not be possible, stalling the process. This is called back-pressure, and buffers can help avoiding it.

In the case of elastic circuits, buffers are placed in two situations, to avoid the mentioned problems of a module stalling others due to needing more cycles to perform its function. These situations are to cut cycles, and to cut paths with a bigger delay than the clock period.

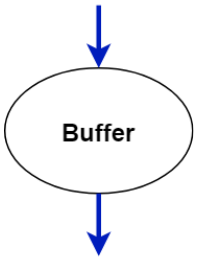

Figure 12: Buffer module (Source: own compilation)

4.3. Constant

A constant is a module that holds a constant value that might be needed in some operations. The idea is to permit the creation of constant modules of the different types that LLVM IR permit to have a constant value.
This module has an input and an output port. The output port is the one that transmits the constant value the module carries, whereas the input port is used to receive only the two handshake signals. As constants cannot receive any type of data from other module, and they are orphan nodes, they need to be triggered somehow. The way they are triggered is with the transmission of the handshake control signals, received from some special blocks that only transmit control signals. The idea is to have at each basic block some handshake signals, either placing some block there, or getting it from some predecessor. This control signal will trigger all the constants present in the basic block, and they will transmit its value to the successor modules.

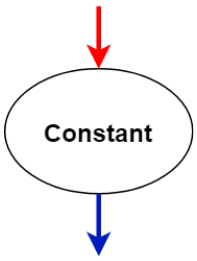

Figure 13: Constant module (Source: own compilation)

4.4. Fork

A fork is a module that copies the data or control signals received through its input port to all its output ports. They can be used to copy data or control signals, and they do not have a limit of output ports. Therefore, they are added every time some port needs to be connected to more than one destination port. There are two types of forks, called Lazy and Eager. A lazy fork is a fork that only outputs the data to the successors when all of them are ready to receive

the data. An eager fork, is more efficient, as it outputs the data to each successor as soon as they are ready to receive new data.

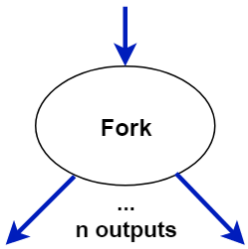

Figure 14: Fork module (Source: own compilation)

4.5. Merge

A merge is a module that transmits the data or control signals received through one of its input ports, to its output port. However, a merge cannot receive data through two input ports at the same time.

Merges are used when some basic block uses a variable that can come from multiple independent predecessors, and only one of them can execute at the same time. As the module that needs the data will not trigger unless the data is present at its input port, we need some mechanism to properly receive the data from whatever predecessor it could comes, avoiding a deadlock due to starving input. If we remember the SSA form, a merge is analogous to the phi functions mentioned there. LLVM automatically places the phi functions in the SSA forms, but we have to also consider the situation when a block can receive data from multiple independent predecessors, but none of them modifies the variable. In this situation we need to ensure the arrival of the data from the execution path taken. Thus, we have to also place merges for all the possible variables that meet these requirements. And these variables are the ones that are alive at the beginning of that basic block, obtained with the live variable analysis. Also, we place merges when we have to receive control signals to activate orphan nodes, from multiple predecessors.

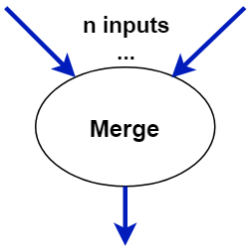

Figure 15: Merge module (Source: own compilation)

4.6. Select

A select is a module that permits the selection of data from one of its two input ports, based in the value of a condition, that receives from another input port. Then it outputs the selected data through its output port. One of its input ports represent the true value of the condition, and the other the false value, but both of them have the same port width. The input port that receives the condition is one of these ports that receives boolean data. We want to mention that the three input ports in a select, are distinguished in the DOT description with a special syntax.

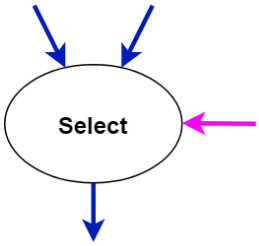

Figure 16: Select module (Source: own compilation)

4.7. Branch

A branch is the reciprocal module of the select. It receives data or control signals at its input port, and depending on the value of a condition it also receives as input, outputs the data through one of its two output ports. Like with the select module, one of the outputs represent the true value, and the other the false value. Also, both the input receiving the condition and the two outputs, have the same special syntax in the DOT file.

We use the branches to transmit the live variables at the end of a basic block to all its successors, as well as transmitting the control signals to them. However, we only create branches when the block has multiple successors, and the branch is conditional. Otherwise, we simply put a channel connecting them. The reason of creating branches for the live variables is the same as with the merges, to ensure the successors to receive data they might need.

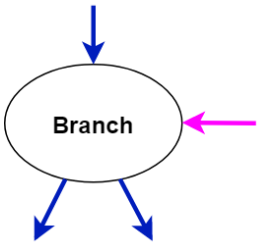

Figure 17: Branch module (Source: own compilation)

4.8. Demux

A demux is a module that has an input port that receives whatever type of data or control signals, and multiple input ports that only receive control signals. The data received through its input port is output through one of its output ports, the same number as the input ports receiving control signals, and with the same width as the input port receiving data. What determines the output port to use is not a condition, but the control signals received through its input ports. But, like a merge, it can only receive control signals through only one port at the same time, and never simultaneously.

This module is rarely used, and in this project it is only used in the way we manage the function call instructions. Besides, in the DOT representation, the order in which the input ports are specified is important, as the first one carries the data and the others carry control signals.

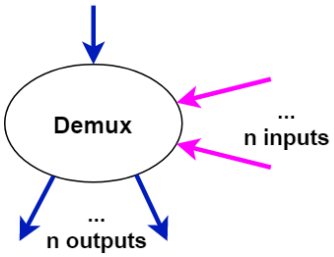

Figure 18: Demux module (Source: own compilation)

4.9. Entry

An entry is a module to implement one of the entries of the graph of a particular function in the LLVM IR. We use this module in the first basic block of a function, that will be in charge of receiving the control signals from the caller, and transmit this control through all the basic blocks of that function. Again, this control will be used to trigger the orphan modules.

We have decided to also implement as entry modules, the arguments of a function. Therefore, it maintains the ports but changing its width, as it will receive and transmit data.

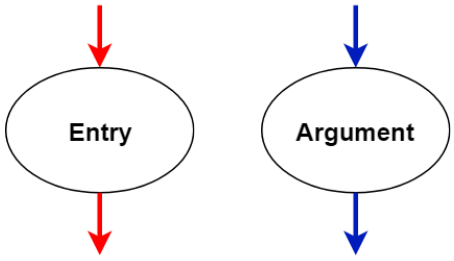

Figure 19: Entry and Argument modules (Source: own compilation)

4.10. Exit

An exit is a module to implement one of the exits of the graph of a particular function in the LLVM IR. We use this block to receive the control signals at those blocks without successors, and return this control to the caller. If there are multiples basic blocks without successors, we place a merge to return the corresponding control.

We have decided to also implement the return instruction as exit modules, as both represent sink modules. However, we will only place a return when the function is not void, placing a merge if there are multiple returns, to gather them all. Also, the ports will carry data instead of control signal.

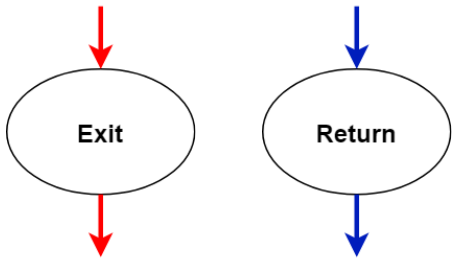

Figure 20: Exit and Return modules (Source: own compilation)

5 Data Flow Graph Generation

In this section we are going to explain how we finally generate the data flow graphs. As we have explained, this part comes after we have already generated the LLVM IR and applied the different passes, and we have already created the class hierarchy to represent the different modules, as well as some classes to represent the basic blocks that will store those modules, and the function that will store those basic blocks. The result of this part will be the final data flow graph, described using the DOT language.

We will implement this part as another pass. This pass will traverse each function inside the LLVM IR, and for each function, and each basic block the function is divided, we will process each instruction. Most of the instructions are a direct translation to one of the defined modules, like the operator module. However, there are instructions that cannot be directly implemented with the available modules. These include those instructions that we have lowered using passes, or those that we will have to create multiple modules to define its behaviour. In addition to create the modules, in order to process the instructions, we will need to connect them through its ports. Besides processing each instruction, we have to perform other steps that involve creating modules with a certain functionality (e.g Fork, Merge and Branch), that we will need

to properly create the graphs, and the latter elastic circuits. We explain below the sequence of steps we have to perform for each function and basic block.

1. In the first basic block of the function we have to process the possible arguments the function can have. These means creating an entry module for each argument.
2. In each basic block we have to ensure the existence of control signals to trigger the possible constants in that block. As we have explained, constants are orphan modules that need to be triggered somehow, to transmit the constant value they hold to its successors.
   - The first basic block will create an entry module that will receive the control signals from the possible caller, and transmit them to the other basic blocks.
   - Basic blocks with multiple predecessors will have to place a merge to receive the signals from the correct one.
   - Basic blocks ending with a conditional branch will have to place a branch module to transmit the signals.
   - Basic blocks without successors will place an exit module to return the control to the caller, creating a merge module if there are multiple basic blocks in the function without successors.
   - In the other situations, a simple channel will be enough to transmit the control signals between the different basic blocks.
3. In each basic block we have to process the live variables found at the beginning of the basic block. As we explained, we have to place a merge module when the data that needs to be used in that block can come from multiple independent predecessors. If the data can only come from one predecessor, the module is not needed, and a simple channel is enough.
4. In all the LLVM IR instructions the constant values are placed directly in the instruction. In the ϕ instructions, that receive values from other basic blocks, happens the same. Therefore, we have to previously save the different constant values that appear in a ϕ instruction, that will belong to a different basic block than the one holding the ϕ. Then, when we process the basic block that should contain that constant values, we have to place a constant module for each of these constant values.
5. Once we have done the previous steps we can process all the instructions in a basic block. Process an instruction consists in creating the module, or the set of modules that are needed to reproduce its behaviour, and connect them with the modules that contain the value of the different operators. To facilitate the connection process, we know at each moment which value is carried by which module, and the concepts of object-oriented programming (OOP) are also a good help. The modules that are connected with a certain module, are stored as references in the origin module of the connection. But thanks to using abstract methods and structures, we do not have to care about the type of module

we are connecting, nor the type of the modules we are connecting with.

6. The return instruction is processed as any other common instruction, creating in this case an exit module, and connecting its operand if the function is not void. However, as we have explained with the control signals, we can have multiple basic blocks with a return instruction, and we will have to place a merge module to gather all the possible returns the function will have, and return the result to the possible caller.

7. The branch instruction only needs to be processed if the branch is conditional, meaning that the basic block has more than one successor. A branch module has to be added for each live variable at the end of that basic block. We have to add a branch in order to properly communicate the data to the path that will execute, depending on the condition. If the basic block has only one successor, or the branch is unconditional, the branch can be omitted and a single channel connecting the modules is enough.

8. The call instruction is the most difficult to generate. For each function defined in the LLVM IR, we may need to add some kind of wrapper, consisting in multiple modules, to manage the different calls that function can receive. However, if the function is called only one time, the wrapper is not needed, and all these modules will be simple connections between the modules from the caller and the called function. Below we show a diagram representing the wrapper of a function that has some arguments, and it is called a few times.

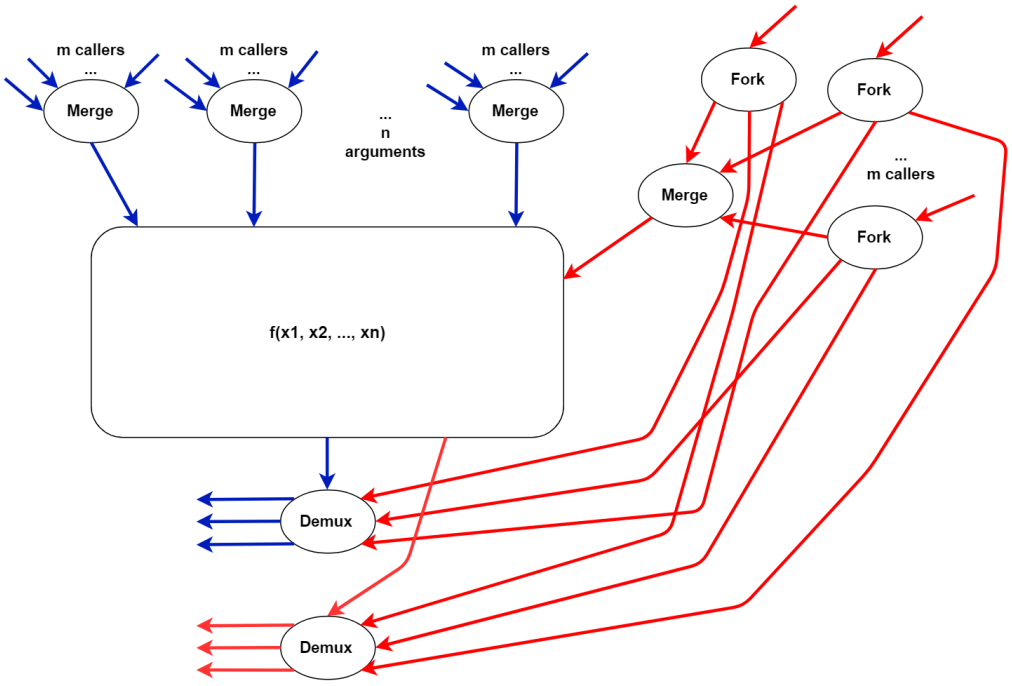

Figure 21: Function call wrapper (Source: own compilation)

When a function is called, the caller must pass all the arguments and some control signal to trigger the possible orphan blocks in that function. As we have said, if the function is

called more than one time we need a wrapper to manage the call. This wrapper consists in some merge modules, fork modules and demux modules.

- We have to place a merge for each argument the function has, as well as one to receive the control signals that will enter the function. These merges will be used to receive data or control signals from the correct caller.
- The control that each caller sends must be split in at least two ways, or three if the function is not void. Therefore, we will add a fork module to replicate the control signals from that caller. One of the outputs of each fork will go to the merge that receives the control, and the others will go to the two needed demux.
- The demux modules will be used to return the control and the possible result to the correct caller. We will have an output port for each of the calls, to return the value through the correct port. If we remember, the output of a demux depends on the control that reaches the demux, only one each time. Therefore, the input control signals will be used to return to the correct caller.

We have decided to create a dummy module to represent each function call. It will be placed in the basic block making the call, and as it will be like any other module, we will connect it with others. Its main purpose is to keep track of the different modules that are involved in that function call, either sending data and control signals, or receiving them.

- If a function is called once, and at the moment we process that call, the function is not processed, we will not have any modules to connect with. However, we have to connect the modules in the caller somehow, otherwise connecting them later will be really complicated. In this situation this dummy block is handy, as we can connect the modules in the caller with it, and later change these connections. We will keep track of the modules that are connected with this dummy block, to facilitate the latter change. The same happens when we have to connect the result and control signals of the function with other modules in the caller, as we do not have them, and again the dummy block can be really useful, as we can use the same methods the other modules use. Again, we will have to modify these connections later, but we are keeping track of the modules that the dummy block is connected with.
- If the function is called multiple times, it does not matter as we have the wrapper, and there are modules that we can connect with. However, we still use the dummy block here to connect it with those modules that will use the result and control signals that the function returns. As the dummy block is placed in the caller basic block, if we have to add forks to replicate the values returned by the function, we will place them in the correct place, and we will not have to worry about placing them correctly.

In both situations, we will need another step to change all the connections once the functions are all processed.

Finally, in each basic block containing calls, we have to synchronize the different control

signals that are present in that block, i.e. the one coming present since the beginning, and those returned by function calls. That is why we have created a type of operator module that will perform a simple synchronization operation of the multiple control signals it will receive, before passing the control to the successor basic blocks.

9. Once each function has been processed, we have to connect the modules that are disconnected, as well as changing those wrongly connected.
    - If a function is called once, we have to connect the modules that we have kept track in the dummy module representing that call, with the argument modules of the function, and the entry module of the first basic block. Also, we have to connect the module containing the returning control signals and the possible result with the modules the dummy module is connected with.
    - If the function is called multiple times, we have to perform the same connections, but in this case the modules are all from the wrapper. The merge modules receiving the arguments as well as the control signals, and the two demux modules returning the result and the control signals.

Finishing with that example presented in the section of the live variable analysis, here we present the generated data flow graph. As it was only a function, the entry and exit modules of that function are disconnected. We have followed with the color convention used to distinguish the different edges. We have placed the modules inside each basic block in a sub-graph environment, in order to distinguish them, as DOT prints them inside a black frame. Also, inside each basic block, we have placed another sub-graph to contain the modules that only communicate handshake signals to trigger orphan modules, but with a red frame to distinguish them.

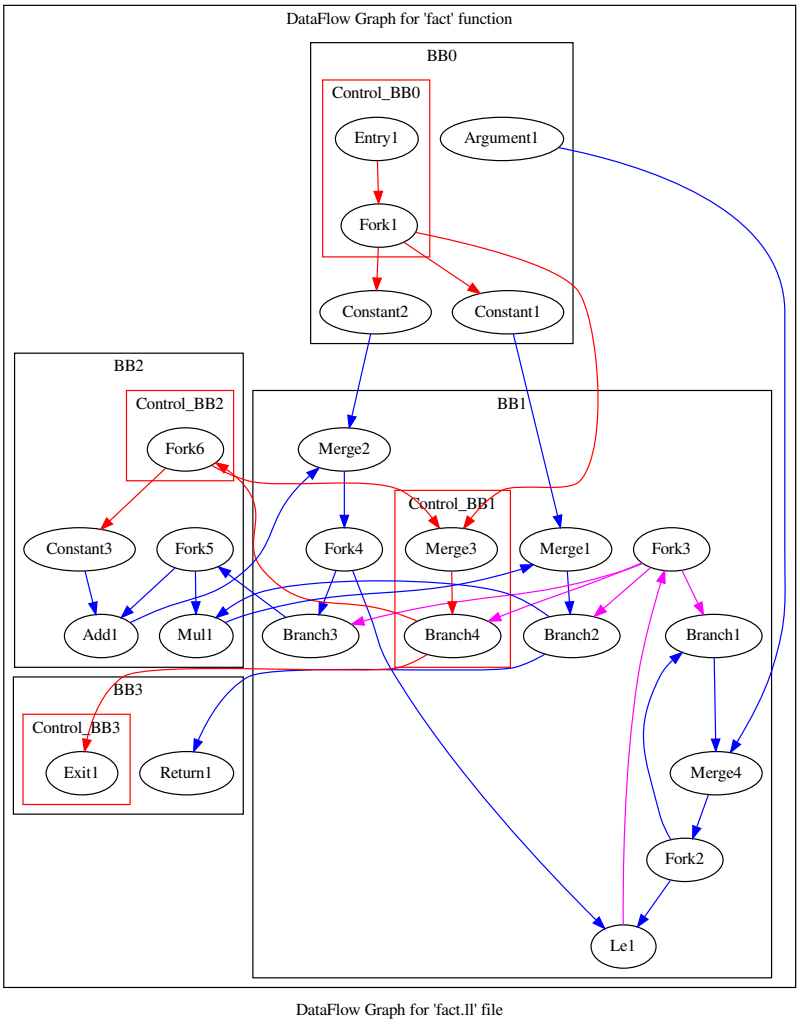

Figure 22: Example of a data flow graph (Source: own compilation)

6 Scope and Methodology

In this section we are going to explain the scope and the objectives we want to reach in this project, as well as the working methodology we will apply to achieve them.

6.1. Scope

This final degree project will go through a series of steps implementing the different concepts and ideas we have presented in the previous sections. We present below the different steps that will divide the implementation of the data flow graphs.

- The first step is translate the high-level source code into the intermediate representation called LLVM IR. As we have explained, LLVM has tools to compile multiple high-level

languages, using that intermediate representation in all of them. For example, *Clang* is used to compile C, C++ and Objective-C.

- With the input code transformed into the LLVM IR, we have to develop the transformation passes we explained before, that are not already defined, that will modify certain instructions. Then, with all the passes created, we have to execute them in the needed sequence in order to transform the LLVM IR.
- Once the LLVM IR is totally transformed, we have to implement the live variable analysis. As we have said, we will also implement it as an analysis pass, permitting its execution with the same tool as the others. Therefore, we have to apply the explained algorithm to each function and basic block, taking into account that the functions are already divided in basic blocks. The results of the analysis will be printed into a text file.
- The next step is defining the different elastic modules with a class hierarchy, as well as defining classes to represent a basic block that will store the modules, and a function that will store basic blocks. In each module we will define the different ports, the block delay, and for some of the modules we will also define the particular attributes they have. The channels will be stored in each origin module, storing a reference to the destiny module and port. The class representing a basic block will have the list of modules in that block, and the class representing a function will have the list of basic blocks.
- Finally, we will create the utility pass that will use everything from the previous steps and create the data flow graphs. As we have said, we will print this graph description in a DOT file with the corresponding DOT syntax.

Due to the time limitations the degree project has, there are some aspects that will not be covered. These include the implementation of the generated data flow graphs into the FPGA. This part implies the programming of all the different elastic modules and channels into the FPGA. Another aspect that might not be dealt is the optimization of the generated graphs. There are situations where some modules can be substituted by simple channels, or situations where some modules of the same type can be fused. The initial agreement was to implement them without optimizing anything. However, we think that it is possible to implement some of these optimizations. What we will try to do is process all the different instructions LLVM IR has. However, there are a really big number, and if problems appear, some of them might not be dealt. In that case, the software will answer by stopping the execution.

6.1.1. Objectives

We have mentioned that we will only perform the first steps of the full solution, and we will leave some aspects uncovered. Therefore, this project will have as its main objective the development of a software that perfectly solves the covered aspects.

As we are developing software with some functionality, we will have to ensure the common features someone would expect.

- Develop a solution using the last available technologies. We have to use the latest version of the LLVM compiler framework and the other programming languages we may use to implement the solution. Otherwise this solution will get obsoleted quickly.
- Offer an intuitive and easy-to-use solution, properly documenting everything. We have to explain in an easy way the different steps to execute the software, as well as explaining the different tools and requisites necessary to execute it. Besides, we have to properly explain how the solution has been implemented.
- As someone might continue this project, we must ensure the most understandable and simple code. We have to write easily readable code, ensuring adequate names to the different variables, structures and functions that appear in the code. In addition, we have to add the needed comments, explaining the functionality of the different parts of the code.
- As one of the competences this project must cover, we must ensure the most efficient and fastest solution. This means that when implementing any of the solution's functions, we have to analyze and decide which are the most efficient features offered by the programming language (i.e. structures, variable types and instructions), and algorithms to address the implementation. We will decide them by determining its time complexity and resources consumption, and use the ones with the best results.
- We must ensure the proper functioning of all the functions this software will offer, by properly testing everything. Each time some function is finished, a battery of test will be applied in order to detect and solve the least odd functioning.

6.1.2. Final Scope

After finishing the project we want to explain the modifications that the scope has suffered, as we have not been able to totally cover all the aspects we considered at the beginning, and some changes have occurred. If the LLVM IR presents some of these uncovered aspects, an assert will be triggered and the execution will stop.

- As we planned at the beginning, covering the implementation of the data flow graphs in the FPGA was totally impossible. Not only due to the big amount of extra hours needed, but also due to the fact that there are some aspects that the director has not completely defined in the implementation of the elastic circuits in the FPGA, like the memory model, that has to be also be programmed in the FPGA, in addition to the different modules.
- In the first specification, the optimization of the data flow graph was not contemplated. However, we have been able to apply some optimizations, whereas others will be applied using some software the professor has, directly applied to the graph description in DOT.

For each variable alive at the beginning of a basic block, a merge module should be placed. The same applies to the set of variables alive at the end of the basic block, where we need to place a branch module. However, merges having only one predecessors, or unconditional branches are useless, and they can be substituted by a simple channel. Placing all of them, even the useless, facilitate a lot the connection of the graph. However it makes the graph more difficult to understand. That is why we have applied these optimizations, and we have simplified a lot the generated graphs. The optimizations we have not applied include the fusion of two forks or two merges connected together. Forks connected together can only appear in the wrapper of a function call, but we do not fuse them to make the graph more understandable. Merges connecting together appear due to the previous simplifications done, but due to lack of time we have not been able to fuse them.

- Another thing we have not considered is the placement of buffers. We have explained that they are essential in the elastic circuits, but the director told us to not worry about them, as we should need specifications of the delays of the different elastic modules, in order to place them, and we do not have those specifications. Therefore, the placement of these buffers is handled by some external software the director has, applied to the graph description in DOT.
- All the modules have some common attributes like the delay of the module, as well as some of them have specific attributes. However, we cannot set these attributes only with the instructions we process. This information is obtained with something similar to the RTL library we seen in the high-level synthesis. Therefore, all these parameters have a default value, but the software is prepared to modify them if needed.
- The instructions that are used to manage dynamic memory have not been handled. As we mentioned, the memory model is not completely defined, but in hardware we do not distinguish the different types of memory, nor we have the system to manage dynamic memory. We do not distinguish the in hardware, nor a system to handle it like in software. Therefore, we do not consider them, only the typical to read and write from memory. We have considered the instruction that reserves space in the stack, but as we do not have a stack either, we can use it to simply reserve space in the memory model of the FPGA.
- The calls to functions defined in external libraries (e.g. math functions like squared root), that are not present in the file we process. We need the definition of each function, in order to correctly generate the circuits. We cannot link these libraries and include the definition in the file containing the LLVM IR, so we cannot permit them.
- Related to function calls, there is a problem with respect to the decided model of the wrapper of a function call. The problem lies within the merges that receives the arguments and control signals. As we explained, a merge should only receive data through one of its input ports, but never two or more simultaneously. And with the defined model, a simpler function that accepts one integer, that is called two times in the same basic block, passing

constant values as the argument, will fail in this requisite the merge has. That is, it will receive data through multiple ports simultaneously, as the constants will be fired at the same time, and the behaviour will be undefined. Therefore, the model of the wrapper that was defined with the director has some mistakes that need to be fixed.

- LLVM IR permits the creation of constants of complex types like arrays and structs. However, we cannot automatically create these recursive types in C++, nor obtaining the values of their elements and fields with the API. Therefore, we have to prohibit them and only accept constants of primitive types.
- Finally the instructions that we present in this list are those that we have not handled due to lack of time. We do not know if they can be represented using elastic modules, as most of them are used to implement an exception-handling system, or they are more complicated versions of other handled instructions, that have some difference in their behaviour. At least, they are not completely essential to perform most of the instructions of a high-level language.

Instruction
invoke
callbr
resume
catchswitch
cleanupret
catchret
unreachable

Instruction
shufflevector
fence
va_arg
landingpad
catchpad
cleanuppad

Table 1: Not yet handled instructions (Source: own compilation).

6.2. Methodology

This project has a short duration, hence we are going to apply an agile methodology in order to succeed. We have decided to apply the Scrum methodology. We are going to set brief periods of one week, where we will set some type of goal that must be reached. However, not for each weak we will fix a big and important goal, but little progress that will permit completing the development satisfactorily and with a good pace.

For those tasks during only a sprint, the objective will be its complete resolution. For those during more than a sprint, the objective of each sprint will be doing a certain amount of work in which the task will be split. For example, one of the programming task will consist in implementing the different components of the data flow graph in order to represent them. As it will last more than a sprint, we will split all the components we must implement in the different sprints, and each sprint's goal will be the implementation of all the assigned components. For those tasks during less than a sprint, we will fix the goal of finishing that task, and all other tasks or parts

of a task split in the same way as above, that can fit in the remaining time.

Setting up these brief sprints, will permit the early detection of programming errors, misconceptions, and other problems that could affect the compliance with the schedule, and even detect a wrong planning.

Besides, the development of the project will be constantly followed by the director. He will receive weekly reports with the progress done that weak and the possible occurred problems and questions that need to be solved. Moreover, extra meetings will be done each three weeks, in order to show the progress so he can really check that everything is following its correct direction, and the project's requirements are fulfilled.

Finally, in order to avoid programming errors and bugs, every week the new code will be properly revised to detect the least problem. Otherwise we will reach a point where we will have a big amount of code lines and detect errors will become difficult.

6.2.1. Monitoring tools

In order to properly monitor the progress of this project, some version control tool will be needed. We have decided to use Git [25] and Github [26], that permit this control of the different versions, as well as keeping all the work in a safe place. Other tools like Trello [27] can be used, in order to keep a good control of all the tasks that need to be performed each week.

Besides the use of these tools, a constant communication with the director will be performed via email or Skype [28].

6.2.2. Validation methods

The director will be following the development at every time, either with the weekly reports or with the different meetings. As he is the one doing research, and the one who proposed the project, he is the one who can validate better the meeting of the requirements and the proper development of the solution. Besides, he will help resolving all the doubts and questions that can appear.

The project will be tested constantly, with some battery of test created by the developer. These tests will not only check the proper functioning of everything, but the efficiency of the solution too. Moreover, the design and the structure of the solution will be also revised constantly.

6.2.3. Final Methodology

After developing the entire project, the used methodology has been maintained and followed. It was quite expected, as it is the one we are most used to work with, and the results have been always positive.

As we explained, we worked with weekly sprints with some objectives to reach. We managed to fulfill those objectives almost all the times, except on one phase where we needed quite extra hours to complete it, increasing the weekly performance during quite the time.

Regarding the communication with the director, he has been following constantly all the development. We kept sending him brief reports via email, with the progress done each week. Besides, we did different meetings with him, even though the initially proposed dates were not always followed. Also, he gave fast feedback and constantly helped with the doubts and problems.

Finally, we used the monitoring tools we decided to use, as well as constantly testing the correctness of our program with tests we created ourselves.

7 Planning

This project will last five months, starting on February 2nd, and ending at most on July 5th. The end date will depend on the project's defense, between the 1st and the 5th of July. Each week will have a dedication of 35 hours (i.e. 5 hours per day), with the possibility of increasing to a maximum of 40 hours if needed. Therefore, the total dedication of the project will be a number between 600 and 700 hours.

7.1. Tasks Description

We will divide the project in different phases, and each phase will have its own tasks. As we will use an agile methodology, the weekly sprints will have as its goal finishing some tasks if they need one week or less, and making the maximum progress in those needing several weeks.

As there is only one person working in the project, we are not going to consider tasks executing in parallel. Only meetings with the director and writing the documentation can overlap a task. Besides, most of the tasks will depend on the previous one, resulting in a sequential execution.

For those phases consisting in implementing code with a certain functionality, we will divide them in the same tasks, i.e. analyze the requirements, design the solution, implement the solution and test the functionality and efficiency of the solution.

7.1.1. Project Launch

In the first phase we will prepare the working environment, and install and learn how to use the needed software tools. After that, we are going to research and read all the needed articles to properly understand the different concepts behind the solution we are trying to develop, as well as looking for other approaches that other people has tried before.

7.1.2. Project Management (GEP)

This phase will include the compulsory course about project management (GEP) that we have to complete at the beginning of the project's life, during the first month. The tasks done in this phase will consist in the four deliveries and the presentation performed in this course. During this phase we will decide some important aspects that will affect the rest of the development. These aspects include:

- Contextualize the project.
- Research the state of the art.
- Define the reach and the goals.
- Plan an appropriate schedule to follow, taking into account the needed resources and the alternatives to deal with delays.
- Define all the costs involved and create a budget.
- Analyze the different sustainability aspects.

In order to properly carry out this phase, the autonomous learning done in the previous phase will be essential, as it can reduce the number of hours reading articles during the fixed delivery periods this course has. Besides, we can consider that all the next phases will depend on this phase, as it is here where we define some aspects like the reach, the objectives, and the planning.

7.1.3. LLVM IR Generation and Processing

This phase will consist in generating the LLVM intermediate representation from the initial high-level code, and then apply some transformations that will be needed, as well as gather certain information from it. The tools used to translate the initial code are the different front ends that can be generated using LLVM (e.g. Clang for C, C++ and Objective C). Then to apply the transformations we have to use the LLVM tool used to apply optimizations to the LLVM IR.

Some of these transformations are already defined. But there are others, that will transform a particular instruction that we will need to define. Besides these transformations, the important part of this phase is the computing of the live variable analysis. An iterative algorithm used to resolve in different points of the program which variables' value will be needed before overwriting

them. This analysis is performed in the optimization phase of a compiler. However, we will use it to add certain nodes in the data flow graphs that are needed to ensure correctness.

7.1.4. Data Flow Graphs Components Definition

This phase will define the graph components that will describe the elastic circuits we want to generate. The nodes in these graphs will represent functional units in the elastic circuits, implementing for example arithmetic and logic operations. The edges will represent transmission of data between those functional units.

We will use the DOT language to describe those graphs. As DOT permits adding custom attributes to both nodes and edges, in the form of pairs of name and value, we can perfectly use the same syntax the language has and only add the needed attributes. For example, each node will have an attribute to indicate its type (e.g. operator), a list describing its input and output ports, and the delay of the block to perform its function. We will have some types that will have exclusive attributes. In the edges we will only have to indicate the ports of the nodes that are connecting, besides indicating the nodes we are connecting.

In order to create this description in DOT, we will create a class hierarchy in C++ to create and store the nodes and edges with their features. With this class system we will take profit of the concepts of Object-oriented programming (OOP) like inheritance and polymorphism, and avoid the necessity of distinguishing the nodes we connect together. Also, we will add some methods to translate the different nodes and edges, stored as class instances, into the DOT language with the custom attributes we will add. With these methods we will create the graph description into a DOT file.

7.1.5. LLVM IR Translation into Data Flow Graphs

This phase will use the results of the previous two phases, and will perform the translation of the LLVM IR code into the final data flow graphs. This translation will consist in traversing the different instructions inside each function, and create the corresponding nodes for each instruction, connecting them as needed. In some instructions a node will be enough, but in other cases it may need multiple nodes and connections to properly implement that instruction. Also, some of these nodes will be special nodes that will be added in certain situations to correctly create the graphs, and ensure a proper circuit description. For example, using the results of the live variable analysis, some nodes will be added.

As we have explained, these nodes will be generated as class instances of a class hierarchy, where we will store all the needed information of both nodes and edges. Once we have all these instances created, we will need to use the mentioned methods to auto-generate the file with the final data flow graph, described with the DOT language.

7.1.6. Documentation and Defense

This will be the last phase of the project, performed during the last month, when all the previous phases are finished. Here, the documentation developed during all the project will be properly revised and finished. It will contain some annexes with a user manual of the developed code, as well as examples of the obtained results. Besides, this phase will include the fully preparation and exposition of the project's defense.

7.1.7. Communication with the Director

As we have explained in the methodology, we will send weekly reports to the director with the goals achieved that week. These reports will not be included as tasks, but we will include all the meetings that we will do every three weeks, as well as the compulsory on May 2nd.

7.2. Resources

7.2.1. Human Resources

This set only includes the developer explained in the stakeholders, doing all the different tasks with a dedication of 35 hours per week, and with a possible increase to 40.

7.2.2. Material Resources

This set involves the different material resources that we will need, i.e., hardware and software resources. The hardware resources only include the laptop needed to develop the entire project. It is an ASUS TP550L, 8Gb of RAM memory, processor Intel Core i7 5500U, a native Windows 8.1 operative system, and a virtual Ubuntu 18.04. The software resources are all free, and they are presented in the table below.

Resource	Purpose
Visual Studio Code [21]	Integrated development environment (IDE) used to program
LLVM Compiler Framework	Front end to compile the input code and generate the LLVM IR
GraphViz	Graph description and visualization application
Git and Github	Management of the code repository
Trello	Plan the different tasks and objectives for each sprint
Webmail (FIB's email) and Skype	Communicate with the director
Overleaf [29]	LaTeX online text processor to create the documentation
Google Slides [30]	Create PowerPoint presentations
Gantter (Trial version) [31]	Create the Gantt chart

Table 2: Software Resources (Source: own compilation).

7.3. Tasks Summary Table

Task	Hours	Dependencies	Resources
1 Project launch	80		
1.1 Set the environment	30	No dependences	Developer, Laptop, Visual Studio Code, Git, LLVM CF, Webmail, Skype
1.2 Autonomous learning	50		
2 Project management	108.1		
2.1 First delivery	40	1 < 2; 2.1 < 2.2; 2.2 < 2.3; 2.3 < 2.4; 2.4 < 2.5; 2.5 < 2.6	Developer, Laptop, Overleaf, Trello, Webmail, Skype, Microsoft PowerPoint Online
2.2 Second delivery	25		
2.3 Third delivery	25		
2.4 Fourth delivery + Presentation Slides	15		
2.5 Prepare the presentation	3		
2.6 Present the presentation	0.1		
3 LLVM IR generation and processing	70		
3.1 Analyze the requirements	5	2 < 3; 1 < 3; 3.1 < 3.2; 3.2 < 3.3; 3.3 < 3.4	Developer, Laptop, Visual Studio Code, Git, LLVM CF, Webmail, Skype, Trello, Overleaf
3.2 Design the solution	20		
3.3 Implement the solution	30		
3.4 Test the solution	15		
4 Data flow graph components definition	135		
4.1 Analyze the requirements	15	1 < 4; 2 < 4; 4.1 < 4.2; 4.2 < 4.3; 4.3 < 4.4	Developer, Laptop, Visual Studio Code, Git, Graphviz, Webmail, Skype, Trello, Overleaf
4.2 Design the solution	20		
4.3 Implement the solution	80		
4.4 Test the solution	20		
5 LLVM IR translation into data flow graphs	140		
5.1 Analyze the requirements	10	1 < 5; 2 < 5; 3 < 5; 4 < 5; 5.1 < 5.2; 5.2 < 5.3; 5.3 < 5.4	Developer, Laptop, Visual Studio Code, Git, LLVM CF, Graphviz, Webmail, Skype, Trello, Overleaf
5.2 Design the solution	20		
5.3 Implement the solution	80		
5.4 Test the solution	30		
6 Documentation and defense	70.5		
6.1 Finish the documentation	50	1 < 6; 2 < 6; 3 < 6; 4 < 6; 5 < 6; 6.1 < 6.2; 6.2 < 6.3	Developer, Laptop, Webmail, Skype, Trello, Overleaf, Microsoft PowerPoint Online
6.2 Prepare the defense	20		
6.3 Defense the project	0.5		
7 Meetings with the director	7		
7.1 Compulsory meeting (50% of the project completed)	1	1 < 7.1; 2 < 7.1; 3 < 7.1; 7.1 will take place during 4.3; Each meeting in 7.2 depend on the previous one and on the current task and phase	Developer, Laptop, Webmail, Skype, Trello
7.2 Regular meetings	6 (1 hour each)		
Total Hours	610.6		

The dependencies are represented as A < B, indicating that task A needs to be finished before starting task B.

Table 3: Summary table with the dedication, time dependencies and resources of each task. (Source: own compilation)

7.4. Initial Gantt Chart

Name	Start	Finish	Hours	Role
Project launch	**02/02/2019**	**02/17/2019**	**80**	
Set the environment	02/02/2019	02/07/2019	30	Programmer
Autonomous learning	02/08/2019	02/17/2019	50	Programmer
Project management (GEP)	**02/18/2019**	**03/29/2019**	**108.1**	
First delivery	02/18/2019	02/26/2019	40	Project Leader
Second delivery	02/27/2019	03/04/2019	25	Project Leader
Third delivery	03/05/2019	03/11/2019	25	Project Leader
Fourth delivery	03/12/2019	03/20/2019	15	Project Leader
Prepare the presentation	03/21/2019	03/26/2019	3	Project Leader
Present the presentation	03/27/2019	03/29/2019	0.1	Project Leader
LLVM IR generation and processing	**03/30/2019**	**04/15/2019**	**70**	
Analyze the requirements	03/30/2019	03/30/2019	5	Analyst
Design the solution	03/31/2019	04/03/2019	20	Designer
Implement the solution	04/04/2019	04/11/2019	30	Programmer
Test the solution	04/12/2019	04/15/2019	15	Tester
Data flow graph components definition	**04/16/2019**	**05/16/2019**	**135**	
Analyze the requirements	04/16/2019	04/18/2019	15	Analyst
Design the solution	04/21/2019	04/23/2019	20	Designer
Implement the solution	04/24/2019	05/11/2019	80	Programmer
Test the solution	05/12/2019	05/16/2019	20	Tester
LLVM IR translation into data flow graphs	**05/17/2019**	**06/16/2019**	**140**	
Analyze the requirements	05/17/2019	05/18/2019	10	Analyst
Design the solution	05/19/2019	05/23/2019	20	Designer
Implement the solution	05/24/2019	06/10/2019	80	Programmer
Test the solution	06/12/2019	06/16/2019	30	Tester
Documentation and defense	**06/17/2019**	**07/05/2019**	**70.5**	
Finish the documentation	06/17/2019	06/26/2019	50	Project Leader
Prepare the defense	06/27/2019	06/30/2019	20	Project Leader
Defense the project	07/01/2019	07/05/2019	0.5	Project Leader
Meetings with the director	**03/15/2019**	**06/28/2019**	**7**	
Compulsory meeting (50% of the project completed)	05/02/2019	05/02/2019	1	Project Leader
Regular meeting 1	03/15/2019	03/15/2019	1	Project Leader
Regular meeting 2	04/05/2019	04/05/2019	1	Project Leader
Regular meeting 3	04/26/2019	04/26/2019	1	Project Leader
Regular meeting 4	05/17/2019	05/17/2019	1	Project Leader
Regular meeting 5	06/07/2019	06/07/2019	1	Project Leader
Regular meeting 6	06/28/2019	06/28/2019	1	Project Leader

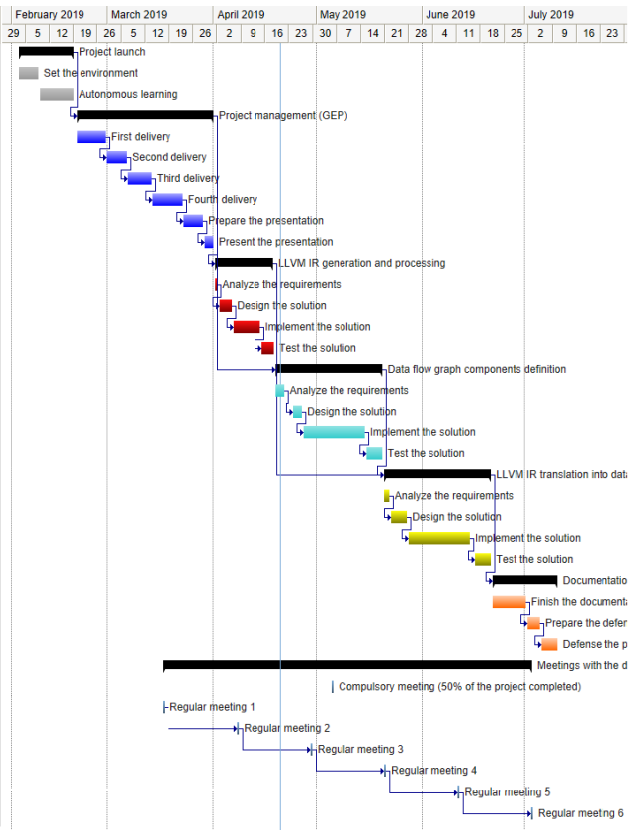

Figure 23: Initial Gantt chart, created with Gantter (Source: own compilation)

7.5. Action Plan and Valuation of Alternatives

As explained before, the brief duration of the project will force some aspects to remain uncovered. However, we will try to not reduce them even more and obey the deadline. Besides, the agile methodology will help with the early detection of problems with the initial planning, and permit their solving as soon as possible.

If a task ends before planned, the next one will start immediately, saving hour for other tasks that could get need more time than initially planned, affecting the development.

If a task needs more time than initially expected, we will have to consider different alternatives to solve the problem without affecting the deadline and the requirements. The only tasks where this situation could happen are the programming-related ones. The others have a fixed period or are easy enough to complete in the planned time. Therefore, we only consider the tasks in phases three, four and five. In particular, we could discard the analysis of the requirements because is the easiest and simplest among the four, as there are not many requirements in each phase. The other three are the most important ones, as they will have more impact in determining the result in each of these phases, and we consider that they can be source of delays.

Besides, among these three tasks that can need more time than planned, the implementation will be the most probably to delay, followed by the testing, and finally the design. Below we present the measures we will apply in this situation, presented in the order they will be applied.

1. **Give broad periods surpassing the estimated duration to complete the task.**
   The estimated dedication for each of those mentioned tasks should be enough, if we do not consider delays. But, if we detect an inappropriate pace in a sprint by not completing its goal, at least exists the possibility of spending more hours (increasing or not the weekly performance) in that task, and reach its finalization date without delaying the start of others.

2. **Re-assign the given spare hours**
   If the first measure is not enough, means that we need more time than planned in some task. Therefore, this measure will consist in redistributing the extra hours assigned to the last three tasks of the programming phases, giving some of them to the delayed task, and redistribute the rest again.

3. **Shorten tasks or combine them together.**
   If the second one does not work either, we will have to shorten tasks, or combine some of them. Below we present this measure, again applied in the presented order.
   3.1. **Combine the analysis and design tasks, performing them together within the time assigned to the design.**
   3.2. **Combine the analysis, design and implementation, performing them together within the time assigned to the implementation.**
   3.3. **Reduce the time assigned to the testing tasks.**
   In phases three and four we can decrease to a minimum of ten hours, and in phase five to a minimum of twenty hours. The five phase combines the two previous ones, so we need more time to test everything assembled. We apply it after modifying the implementation and design because we consider that it can be more important than those two, as in these tasks we ensure the correct functioning of everything.
   3.4. **Reduce the time assigned to finalize the documentation.**
   We will try to write it during the whole development, but we cannot ensure its fully compliance. For this reason, this will be the last measure to apply, and we will reduce a maximum of 20 hours, depending in the state of the documentation at that moment.

Besides all these presented measures, in all the tasks that can delay, we can consider an increase in the daily performance. We have considered before adding one hour per day, but we could increase it even more without a fixed limit, and during the needed time.

Whatever deviation with respect to the initial plan will affect both human and material resources, as the human resources will work extra or less hours and use the material resources (i.e. hardware and general expenses) to do it.

Finally, we want to mention that with the initial working performance and the given duration, completing the project within the initial plan is perfectly feasible. We have not considered any measure that involves reducing the objectives or the scope. The aspects we are covering at the minimum we should cover in this particular project. That is why, we should not decrease them. However, it could happen that some aspects in LLVM IR cannot be currently translated into a hardware circuit, and we have to left them uncovered at the moment.

7.6. Deviations and Final Plan

During the GEP course, we planned the whole development of the project in the way we have explained in the previous sections, dividing it in different phases and tasks, estimating the duration of each task. However, the followed plan differs with the initial plan. Below we explain the modifications made on each phase, taking into account that the first phase ended before starting GEP, and that the GEP phase had a strict calendar to follow.

- As we have mentioned, the GEP course had a strict calendar to follow, and delaying those tasks was impossible. However, the day of the presentation took place before it should have happened.
  This phase ended needing 5 extra hours in the fourth delivery, to correct the errors made in the second, and other 5 to prepare the presentation. However, these extra hours did not affect the planning, as there were enough margin to finish each of them.

- The third phase started quite before the initially planned. The first regular meeting with the director was planned in the middle of the GEP phase. Therefore, we decided to work extra daily hours to make the maximum progress and show something to him. The day of that meeting, the implementation task of the third phase was more or less finished, but there were things that needed to be fixed and others improved. However, the work with that phase stopped there until the end of the GEP course, when it was resumed. This phase ended at the end of the month, half month before the initially planned.
  The hour estimation was more or less accurate, as the analysis was simpler than expected, saving 3 hours, but needing 5 more in the implementation to fix the mistakes made during the first regular meeting. Again, these extra hours did not affect the plan. This was because of that measure to deal with delays, consisting in giving spare hours to some tasks, surpassing the estimated duration.

- At the beginning of April the fourth phase started, finishing it more than a week before ending the month.
  This phase was not that accurately planned. In the end, this phase was easier than expected, and it needed 38 hours less than initially planned, divided among the four tasks. Although there were some aspects that we needed to clarify with the director, both the requirements and the design were quite simple, only needing time to do the

implementation. For this reason, the test was also simplified.

- The fifth phase started one week after ending the fourth. This was because we had to write a report about the situation of the development, once the development passed the midpoint. This phase took an entire month to be completed.
  Comparing it with the initial Gantt, it seems that this phase also took less time to be performed. However, the real hour consumption reflects an important increase and a bad estimation. Although the analysis, design and test needed less hours, the implementation needed 46 extra hours. As the documentation was put aside a bit for some time, we increased the weekly performance during the whole month to finish this phase faster and have enough time later to finish the documentation. That is why we managed to end this phase faster.

- Thanks to making an effort to finish the programming as soon as possible, we resumed the writing more than half a month before the initially planned. At that point we had written more or less the 50%. Besides, the end date of that task was advanced, because it was compulsory to deliver this document one week before the project's defense. This also advances the preparation of the defense, having more time to spend in that task, as the date of the defense remains the same.
  The hour estimation in this phase has suffered an increase of sixteen hours. One extra hour to finish the documentation, and fifteen to prepare the defense, as we have more time than initially planned and we want to use it.

- The dates of the regular meetings have suffered some modifications than initially planned. The first regular meeting as well as the compulsory meeting took place as planned. The second was delayed a week, because the second phase barely started, and there was not enough progress to show. The third one, following the rule of one meeting each three weeks, overlapped the compulsory. For this reason, we decided to delete one of them due to lack of time, and re-arrange the others to fill the rest of the development.

7.7. Final Gantt Chart

Name	Start	Finish	Hours	Role
Project launch	**02/02/2019**	**02/17/2019**	**80**	
Set the environment	02/02/2019	02/07/2019	30	Programmer
Autonomous learning	02/08/2019	02/17/2019	50	Programmer
Project management (GEP)	**02/18/2019**	**03/26/2019**	**120.1**	
First delivery	02/18/2019	02/26/2019	40	Project Leader
Second delivery	02/27/2019	03/04/2019	25	Project Leader
Third delivery	03/05/2019	03/11/2019	25	Project Leader
Fourth delivery	03/12/2019	03/20/2019	20	Project Leader
Prepare the presentation	03/21/2019	03/25/2019	10	Project Leader
Present the presentation	03/26/2019	03/26/2019	0.1	Project Leader
LLVM IR generation and processing	**03/05/2019**	**03/31/2019**	**72**	
Analyze the requirements	03/05/2019	03/05/2019	2	Analyst
Design the solution	03/05/2019	03/09/2019	20	Designer
Implement the solution	03/10/2019	03/28/2019	35	Programmer
Test the solution	03/29/2019	03/31/2019	15	Tester
Data flow graph components definition	**04/01/2019**	**04/21/2019**	**97**	
Analyze the requirements	04/01/2019	04/01/2019	5	Analyst
Design the solution	04/02/2019	04/04/2019	15	Designer
Implement the solution	04/05/2019	04/19/2019	67	Programmer
Test the solution	04/20/2019	04/21/2019	10	Tester
LLVM IR translation into data flow graphs	**04/27/2019**	**05/27/2019**	**174**	
Analyze the requirements	04/27/2019	04/27/2019	5	Analyst
Design the solution	04/28/2019	04/30/2019	15	Designer
Implement the solution	04/30/2019	05/23/2019	126	Programmer
Test the solution	05/24/2019	05/27/2019	28	Tester
Documentation and defense	**05/28/2019**	**07/05/2019**	**86.5**	
Finish the documentation	05/28/2019	06/23/2019	51	Project Leader
Prepare the defense	06/24/2019	06/30/2019	35	Project Leader
Defense the project	07/01/2019	07/05/2019	0.5	Project Leader
Meetings with the director	**03/15/2019**	**06/28/2019**	**6**	
Compulsory meeting (50% of the project completed)	05/02/2019	05/02/2019	1	Project Leader
Regular meeting 1	03/15/2019	03/15/2019	1	Project Leader
Regular meeting 2	04/12/2019	04/12/2019	1	Project Leader
Regular meeting 3	05/23/2019	05/23/2019	1	Project Leader
Regular meeting 4	06/17/2019	06/17/2019	1	Project Leader
Regular meeting 5	06/28/2019	06/28/2019	1	Project Leader

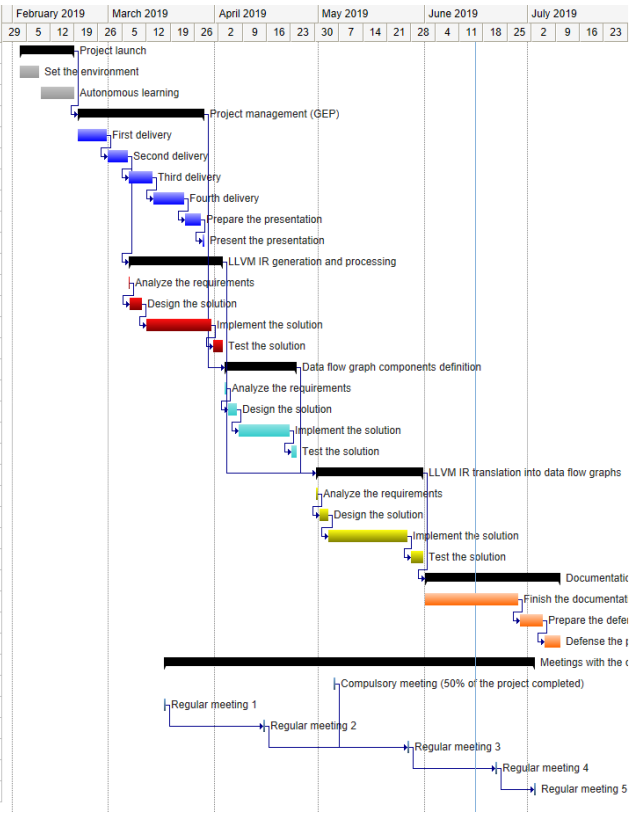

Figure 24: Final Gantt chart, created with Gantter (Source: own compilation)

As we can see in the final Gantt chart, we have been able to end all the tasks and reach the finalization date without any problems. It is true that we needed extra 25 hours with respect what we planned at the beginning. Although, 15 of them are used to prepare the defense better, and we could save them.

Nevertheless, we have to mention that although everything has gone quite good, the initial plan we made was quite inaccurate. All the tasks have suffered deviations, all starting before planning, almost all of them needing more or less hours than planned, and even needing extra hours in tasks we did not considered at the beginning.

In the economic management as well as in the sustainability study, we will measure the impact this final plan has, and compare it with the initial one.

8 Economic Management

8.1. Cost Identification and Estimation

In this section we will identify and estimate the costs of the different resources needed, the general expenses to use some of them, costs associated with possible unexpected events, and finally some contingency costs to deal with possible deviations. All these costs will form the final budget.

8.1.1. Human Resources

As explained before, one person alone will adopt all the involved roles and perform the different tasks. The possible roles we are considering, are the typical ones in a software project. They are presented in the table below, with the total hours, cost per hour, and total cost of each of them. The cost per hour is an approximate value obtained by contrasting the same roles in other degree projects and some web pages.

Role	Hours	Cost per hour (€/h)	Cost (€)
Project leader	185.6	19.80	3674.88
Analyst	30	17.00	510.00
Designer	60	13.63	817.80
Programmer	270	11.30	3051.00
Tester	65	11.30	734.50
Total	610.6		8788.18

Table 4: Roles involved in the project (Source: own compilation based on market prices)

Below, there is another table where the human costs are further detailed, divided for each of the tasks we have presented in the planning.

Task	Hours	Roles	Cost (€)
1 Project launch	80		904
1.1 Set the environment	30	Programmer	339
1.2 Autonomous learning	50	Programmer	565
2 Project management	108.1		2140.38
2.1 First delivery	40	Project leader	792.00
2.2 Second delivery	25	Project leader	495.00
2.3 Third delivery	25	Project leader	495.00
2.4 Fourth delivery + Presentation Slides	15	Project leader	297.00
2.5 Prepare the presentation	3	Project leader	59.40
2.6 Present the presentation	0.1	Project leader	1.98
3 LLVM IR generation and processing	70		866.10
3.1 Analyze the requirements	5	Analyst	85.00
3.2 Design the solution	20	Designer	272.60
3.3 Implement the solution	30	Programmer	339.00
3.4 Test the solution	15	Tester	169.50
4 Data flow graph components definition	135		1657.60
4.1 Analyze the requirements	15	Analyst	255.00
4.2 Design the solution	20	Designer	272.60
4.3 Implement the solution	80	Programmer	904.00
4.4 Test the solution	20	Tester	226.00
5 LLVM IR translation into data flow graphs	140		1685.60
5.1 Analyze the requirements	10	Analyst	170.00
5.2 Design the solution	20	Designer	272.60
5.3 Implement the solution	80	Programmer	904.00
5.4 Test the solution	30	Tester	339.00
6 Documentation and defense	70.5		1395.90
6.1 Finish the documentation	50	Project leader	990.00
6.2 Prepare the defense	20	Project leader	396.00
6.3 Defense the project	0.5	Project leader	9.90
7 Meetings with the director	7		138.60
7.1 Compulsory meeting (50% of the project completed)	1	Project leader	19.80
7.2 Regular meetings	6 (1 h/m)	Project leader	118.80
Total	610.6		8788.18

Table 5: Human resources costs detailed at the level of the Gantt tasks and phases (Source: own compilation based on market prices)

8.1.2. Software and Hardware Resources

All the software resources we are going to use are open source, so they are free and no license is needed. The hardware resources include only the before mentioned laptop. The cost of that laptop was 700€ when it was purchased in October 2015. We will calculate its amortization considering a period of 4 years, and a working pace of 8 hours per each day of the week. We will calculate the amortization per worked hour in that period, and then multiply it by the total hours. The calculation is:

$$(700€ * 610.6h)/(4years * 365days/year * 8hours/day) = \mathbf{36.59}€$$

8.1.3. General Expenses

General expenses will include the laptop electricity consumption and the internet access costs. The laptop consumes 65W. Assuming a current kWh price of 0.13€, the calculation is:

$$0.065kW * 610.6h * 0.13€/kWh = \mathbf{5.16€}$$

The Internet access amortization is calculated as the amortization per worked hour in a year, considering the same working period of 8 hours per day, and then multiply it by the total duration of the project. Considering a monthly fee of 40€, we obtain the following calculation:

$$(40€/month * 12months/year * 610.6h)/(365days/year * 8hours/day) = \mathbf{100.37€}$$

Adding them it results in a cost of **105.53€**.

8.1.4. Unexpected Events

We will consider two unexpected events. The first one will be the break of the laptop used to develop everything. The second one will be the mentioned delays in the programming phases. The cost of each of them will be multiplied by its probability of occurrence.

If the laptop breaks, it will be repaired if possible, or replaced otherwise. We will consider the higher cost of replacing it by the same model, and with a probability of 10% as it works perfectly but it is quite old. Therefore, the cost will be **70€**.

The worst scenario of planning deviation is when all the last tree tasks of a programming phase need more hours than estimated, but they can still reach its finalization date, and no re-assignment of hours and periods is needed. Otherwise, it would reduce the costs as the hours of project leader, analyst or designer would be probably substituted for programming hours or for other type with lower cost per hour. We will add 5 testing hours in the three phases, 25 programming hours and 10 design hours in phases four and five, and only 15 programming hours and 5 design hours in phase three. A probability of 20% assigned to phases four and five, as they have similar complexity, and 15% to phase three as it is easier. Using the same formulas and considerations as before, and applying the probability of occurrence, the human costs of phases four or five ascend to **95.06€**, and **44.12€** in phase three. The hardware amortization and general expenses ascend all together to **1.86€** in phases four or five, and to **0.87€** in phase three. Adding all of them it results in **238.83€**

Adding the cost of the two unexpected events results in a total cost of **308.83€**.

8.1.5. Contingency

The contingency costs will suppose the 15% of the addition of the direct costs (i.e. human resources) and indirect costs (i.e. hardware and software resources and the general expenses),

as the budget is more or less detailed. Therefore, the contingency costs will amount to:

$$(8788.18€ + 36.59€ + 105.53€) * 0.15 = \mathbf{1339.54€}$$

8.2. Initial Budget

After identifying and estimating the different costs we have to consider in the project, the budget is presented in the table below.

Type	Amount (€)
Human resources	8788.18
Software and hardware resources	36.59
General expenses	105.53
Unexpected events	308.83
Contingency	1339.54
Total	10578.67

Table 6: Initial budget (Source: own compilation based on market prices)

8.3. Management Control

In order to properly control the budget, at the end of each task we will update the budget with the real hours and costs of the different human and hardware resources, general expenses, and unexpected events associated to that task. We will compare them with the estimated costs and hours, in order to detect deviations as soon as possible, and easily detect their origin. Besides, we will be able to adjust the estimation in the next tasks and phases, and make it more exact.

The metrics we are going to use are both the cost deviation and consumption deviation of those resources, general expenses and unexpected events, in order to know if a deviation is due to a difference in worked hours or it is due to a difference in the cost per hour. We will add the two metrics in order to obtain the total deviation of each of these resources, expenses, and unexpected events in each task. The formulas to these metrics are presented below.

$$Cost\ Deviation = (estimatedcost - real\ cost) * realhours$$
$$Consumption\ Deviation = (estimatedhours - real\ hours) * real\ cost$$
$$Total\ Deviation = Cost\ Deviation + Consumption\ Deviation$$

The possible deviations in the budget will be due to the unexpected events explained before, and the most possible will be the delay in tasks leading to more working hours. However, we can allow some deviation, as we have taken into consideration the costs of those unexpected events, as well as contingency costs to cover deviations in the final budget.

8.4. Final Budget

Once the project has fully developed, we present the final budget, taking into account all the deviations in the planning. Also, we have been using those metrics to properly control the deviations in the budget, that we have explained before. However, these deviations have obviously been in consumption, as we are not considering changes in the costs per hour. In the end we have needed 25 extra hours to finish all the task. The problem is that among these hours, most of them belong to the role of project leader, the most expensive. We did not consider this possibility, as we ended having more time to prepare the defense than planned and we have decided to use it.

Type	Amount (€)
Human resources	9174.28
Software and hardware resources	38.09
General expenses	109.85
Total	9322.22

Table 7: Final budget (Source: own compilation based on market prices)

If we compare the initial and final budget, we can see how with the costs assigned to the unexpected events we could almost cover the deviations with respect the initial budget. If we add the contingency costs, we can fully cover them. This is a good result, as we are not presenting a budget that surpass the initial one, but these results also show the different mistakes we have done when planning the project.

9 Sustainability

9.1. Sustainability Auto-evaluation

After answering the survey proposed during the GEP course, that can be found in `goo.gl/kWLMLE`, I have come to realize that there are lots of sustainability concepts that I do not apply very often, like the creation of a full budget with contingency and unexpected costs. Besides, there are concepts I have never heard about before, like the deontological principles, or concepts that I did not thought of applying in the development of a project, like diversity or equity.

During these years in the degree, in all the projects I have developed, I never considered directly the three dimensions of sustainability, as it was not necessary to succeed, nor a requisite to take into account. When I develop some software in a project, I always try to develop an innovative and intuitive solution, easy to use, re-usable, that takes into account the resources consumption, uses the newest technologies, and satisfies all the necessities and users. These aspects are quite related with the three dimensions, but I never measure them nor make a proper evaluation. In

fact, this was the first time I computed the amortization costs of the hardware used, or its consumption in kWh, for the sake of analyzing the impact they have.

However, I am quite confident that I can perform an adequate sustainability analysis, using metrics to measure its impact. I control the environmental and economical dimension quite well and I understand all its different concepts and metrics. However, the social dimension is the one I control less. I always try to satisfy the users who will benefit from what I develop, but I do not know how to proper measure the impact. Also, there are concepts in the survey related to that dimension, some of them mentioned at the beginning, that I do not know how to properly apply. Besides, I lack practice with project management, as I usually prefer to work alone, and in the past situations a simple planning was enough. The same applies with collaborative working tools, I know them but I am not too much familiar. My experiences with team working were quite bad when I had to work with people I did not know, and in the end I preferred to work alone and avoid problems.

As a conclusion, I want to say that making this project will change the way I will face future projects with similar complexity and importance. And I hope the degree puts more effort in teaching more about these sustainability concepts, or at least make the students analyze their impact a bit, making the students more comfortable when working with them.

9.2. Sustainability Matrix

As we have explained before, this final degree project covers only the first steps of a research project proposed by the advisor. As it is a research project, the main goal is not selling anything nor making profit of the result, once it is totally finished, but analyze if the approach taken is correct and could be further studied, paving the way for something bigger with great benefits. Especially, if the possibility of this approach having negative results and being discarded could exists. And the possibility of discarding this approach is the biggest risk at all, but we cannot currently predict the probability of this happening, as the solution is not yet completed.

Therefore, as it is part of a research, the concept of useful life is difficult to apply, even more if we only take into account the work we have done in this degree project. For this reason, we are going to consider as useful life the process of analyzing and studying the obtained results in time and power efficiency, as well as the improvements that the approach could need. And this useful life could end with the discard of the approach or the success in solving the problem.

Below we present the sustainability matrix, whose cells will be explained in the following subsections. The first column ranges from 0 to 10, the second from 0 to 20, and the third from -20 to 0. In the three cases the first value represents that the particular aspect is totally unsustainable, and the right value fully sustainable. Due to the part we have done cannot be very much used alone, the punctuation we give also include the complete solution.

	PPP	Useful Life	Risks
Economic	6	13	-10
Environmental	8	16	-10
Social	7	14	-2
Sustainability Degree	21	43	-22
	42		

Table 8: Sustainability Matrix (Source: own compilation)

9.3. Environmental Dimension

9.3.1. Project Put into Production

During the GEP course we made an estimation of the environmental impact the development would have, considering the initial plan and the estimated duration. Among all the resources, only the laptop, that consumes 65W, has some impact. The initial plan estimated a duration of 610.6h, meaning a total consumption of 0.065W*610.6h = **39.69kWh**. Taking into account the unexpected events (i.e. a maximum of 105 extra hours due to delays) we considered there, the consumption would suffer an increase of **6.82kWh**.

Considering the followed plan, that has supposed a development of 635.6 hours, the consumption should be **41.31kWh**. However, we have taken advantage of using a laptop and its battery life, and we kept it disconnected from the electricity 1.5 hours each day. Considering that we have worked 5 hours almost each day, we can obtain an approximated reduction of **12.39kWh**, not despicable at all. As the approach we are taking is quite a novelty, reusing resources, especially open source software, has not been possible. At least, using the LLVM compiler framework has avoided the necessity of spending hours creating front ends to compile different high-level languages. In addition to the software tools, we only need the laptop, so considering the reuse of more resources is not possible.

If we were to consider the part of the solution that we have not covered in this degree project, the development would suppose a bigger impact. In addition to the extra hours to finish the uncovered aspects, we would have to add the consumption of the FPGA used to test the solution. The computer used to finish the solution could perfectly be the same used here, or with the similar specs and consumption. The consumption of the FPGA would depend on the model used, and on the way it is programmed. As its hardware is programmable, it does not consumes always the same, supposing an advantage regarding CPUs and GPUs that have a fixed circuitry and a minimum power consumption.

9.3.2. Useful Life

As mentioned, the useful life of this project will be the further research, study and upgrades that the approach could need in order to solve the problem. However, if the results are not promising enough, this approach could end rejected.

Therefore, the environmental impact of using the solution would be the consumption of the computer in charge of executing the software to synthesize high-level code into elastic circuits, and the consumption of the FPGA to run the generated circuit. As with the development, the consumption of the computer could be similar to the one we have used, and the same happens to the FPGA, that would depend on the model and the circuit implemented in there.

However, as mentioned in the objectives, one of the main goals of the complete solution is to reduce the power consumption when executing programs. Therefore, if the project could achieve good results, we could reduce the consumption when executing programs, compared to the ones we have with the commonly used CPUs and GPUs.

Finally, if we were to compare the impact this approach would have with respect to those presented in the state of the art (e.g. [1] and [8]), it would be probably worse in this approach. The goals and technologies used are the same, the only change is in the type of circuits that are implemented in the FPGA, or the intermediate representation used. Therefore, the benefits obtained with this solution should be similar to those achieved in other approaches. However, the impact of the development process will be higher, because we have probably needed more hours to develop the aspects we have covered, as we were only person developing everything who was not an expert.

9.3.3. Risks

As reducing the power consumption when executing programs is one of the goals the research seeks, the only risk the approach could undergo, it is the failure of achieving a good reduction with respect the commonly CPUs and GPUs. It could achieve some reduction, but less than expected, not being enough to counter the disadvantages in costs and program difficulty the FPGA has with respect the other two. Or it could not achieve a reduction at all, and even an increase with respect the current used technologies. This would mean modifications in the approach, or even its rejection.

9.4. Economic Dimension

9.4.1. Project Put into Production

As with the environmental dimension, during the GEP course we made an estimation of the budget that the development of the covered aspects would undergo, with all the possible costs. Those costs were calculated using only real and present costs per hour, fees and amortization periods. Also, we considered the most realistic probability of the unexpected events to happen, as well as an appropriate contingency percentage for the presented level of detail. As it is detailed in the economic management, the total estimated budget was **10578.67€**.

Now that the project has ended, the final budget has been **9322.22€**. As we have explained before, we have needed 25 extra hours, divided in the role of project leader and programmer. Nevertheless, we have been able to cover this increase with the contingency and the cost of unexpected events, so everything is correct. For this reason, trying to save some money has not been possible, as we have deliberately used 15 of those 25 hours to better prepare the project's defense.

If we were to consider the aspects we have not covered, the budget should include more hours to develop and document the uncovered aspects (i.e. more hours in all the roles), as well as the costs of the FPGA to test the solution. It would depend on who continues it, but 200 hours are perfectly feasible. And the costs of the FPGA would depend on the model used, but FPGAs can range from hundred dollars to thousands.

9.4.2. Useful life

Once someone finishes the work we have left uncovered, other people will be needed to study the obtained results in both time and power consumption with respect to the commonly used technologies. Also, if the results were promising, the approach could need modifications and improvements. For this reason, someone who improves the software we have developed in this project, as well as the one connecting with it, that finally implements the elastic circuits in the FPGA, will be also needed. Improving the software would mean improve its time and resource efficiency, its power consumption, and adding more functions if needed. Also, the used FPGA will need maintenance, changing it when needed. It is true that like CPUs and GPUs, FPGAs have a long lifespan that can surpass the 20 years, but they are usually changed due to obsolescence. However, changing an FPGA is some times more expensive than changing a CPU or a GPU, depending on the model used.

If we compare again this approach to those mentioned in the state of the art, the costs to maintain the project, once it is set, would be similar. Again the differences will be in the development costs due to our fault, and for the same reasons that we explained. Therefore,

this solution will not economically improve others. However, as we are doing a research to offer an approach that could solve an unsolved problem, and we are not trying to sell anything, the economic costs to develop the project are not that important, and what really matters are the possible promising results that could be obtained with the complete solution.

9.4.3. Risks

As mentioned before, this is part of a research and the main goal is not obtaining a profitability. Especially, if we consider that the researched topic and the approach used are quite new, and the results could not be optimal. Therefore, as we also mentioned, depending on the conclusions the project achieves, the approach could need changes or end discarded. And this means that the biggest economic risk is the failure of the project itself, rather than any other external situation once the project is set. That is, if the reduction in time and consumption would not be enough to suffice the increased costs of the hardware, it could suppose discarding this approach.

9.5. Social Dimension

9.5.1. Project Put into Production

The realization of this project has been a good help to realize what the development of an important project suppose, not only the programming parts and the difficulties of facing new programming languages and concepts, that I am most used to, but especially the management of the project. I have also learned new concepts related to the project's topic like the functioning of FPGA, its differences with other existing similar hardware, the elastic circuits, and the high-level synthesis.

Probably, most of these concepts will only serve as more knowledge, but not as something I will constantly apply. It will depend on the path I take after ending here. However, I can ensure that all the concepts related to project management as well as project sustainability will be useful, and they have changed the way I should handle this type of projects.

9.5.2. Useful Life

As we explained in the stakeholders, the main beneficiary will be the director of this project. He is the one who proposed this project, and the one doing research about the topic. As we have not completely finished it, he will have to look for someone who can finish. Hence, the work we have done could be useful to the person who finishes it.

If the results obtained with this approach, once it is finished and analyzed, are promising, we could also consider as beneficiaries other researchers on the topic. They could use this approach with different purposes like using it as basis to develop their own, or compare their results with the ones obtained with this approach.

Then, we could also consider other beneficiaries, but only if the approach really solves the problem. In that situation we could include hardware and software engineers, and even common people. We could help software engineers who are not used to hardware programming, to synthesize their code into elastic circuits, accelerating the execution of their programs and algorithms. We could help hardware engineers, facilitating their process of designing and verifying efficient hardware with an increase complexity. Finally, common users could also benefit in an indirect and transparent way, when executing programs or applications. In that case, start using other technologies besides the commonly used, could affect those who produce the new and old ones.

But, this situation happening is not that easy, as in the end the complete solution could not solve the problem and this approach ending discarded. Or in case it has positive results, it will need a lot of further study to improve and update this first approach, and finally reach that point.

However, we still think that this project should be developed. We want to solve an unsolved problem with an alternative that others have not considered, that could make a difference. That is why it is a research. And with the work we have done in this final degree project, we are paving the way for someone to finish it and valuate the effect of this solution.

9.5.3. Risks

The only situation that could negatively affect some sector would be the complete success of this approach to solve the problem, that could lead to the growth of using better technologies like FGPAs and decrease the use of the old ones. This would affect the producers specialized on manufacturing the old types. However, the probability of this situation happening is currently really low.

10 Used Knowledge and Worked Competences

In this section we are going to explain the knowledge that we have used in this project, that has been learned during the different courses taken in the degree. We will also explain the reasons why this project belongs to the specialization taken, and the technical competences of that particular specialization that we needed to work and achieve during the whole project.

10.1. Used Knowledge

As mentioned in the contextualization, this project belongs to the field of compilers, whose concepts were learned in the Computer Science specialization. For this reason, the main knowledge this project integrates belongs to the area of the compilation of programming languages. This knowledge was learned in the compulsory course called Programming Languages where these concepts were introduced, and further explained in the optional course called Compilers, taught by the project's advisor.

These are mainly the concepts that makes this project belong to the specialization of Computer Science. However, there is other knowledge learned in the common courses done during the first years of the degree that we have also applied. These include the programming manners learned during all the different programming courses. In particular, the concepts learned in the course called Data Structures and Algorithms, related to the analysis and consideration of the memory and time consumption in the programs we create. That is, when we decide the best way to program something, selecting the different algorithms, functions and operations to perform, as well as the different data structures and variables to store content. These concepts were also reminded in the course called Algorithmics, that also belongs to the Computer Science specialization. Besides, we have to include the object orientation concepts like polymorphism, inheritance, and abstraction, taught in the course called Programming Projects.

10.2. Worked Competences

10.2.1. CCO1.1

To evaluate the computational complexity of a problem, know the algorithmic strategies which can solve it and recommend, develop and implement the solution which guarantees the best performance according to the established requirements. [In depth]

As one of the main objectives of the research project is to reduce the time and power consumption compared to the execution in CPUs and GPUs, we have to ensure the best time performance and best use of the machine resources in the translation of the high-level code into the data flow graphs. Therefore, this competence has been applied during the complete development.

Using the LLVM C++ API has been mandatory, otherwise parsing the LLVM IR code as strings or some other representation would have been a performance downgrade. Also, developing everything as LLVM transform and analysis passes has been a good strategy, as it has facilitated the traverse of functions and instructions, as well as permitting the easy sequencing of passes,

and using data calculated in other passes. Also, representing the elastic modules as a class hierarchy has been a good strategy to easily connect modules without worrying about the types involved. Besides, we have tried to use the minimum storage as well as using the most efficient structures to store the content needed to properly create the graphs.

10.2.2. CCO1.2

To demonstrate knowledge about the theoretical fundamentals of programming languages and the associated lexical, syntactical and semantic processing techniques and be able to apply them to create, design and process languages. [In depth]

This is the most important competence, as we are developing a compiler to synthesize code written in high-level languages into elastic circuits, and implement them in an FPGA. It is true that we are leaving some of the processes to the LLVM tools, that generate the intermediate representation, and are in charge of ensuring the correctness of the input code. However, we are in charge of processing that intermediate representation, processing all the different instructions and operands, and generate the corresponding elastic modules and channels.

Therefore, we are applying the lexical and syntactical techniques to properly generate the graphs. The LLVM performs the semantic analysis and verifies that everything in the original code is well constructed, as well as generating the intermediate code. And then, we take care of generating code for the DOT language, as well as applying some modifications to the LLVM IR, and optimizations on the generated graphs.

10.2.3. CCO1.3

To define, evaluate and select platforms to develop and produce hardware and software for developing computer applications and services of different complexities. [A little bit]

In this project, the decision of using both LLVM IR and the DOT language was quite established since the beginning. The idea of the approach was to use some intermediate representation for multiple high-level languages, and LLVM was the best option as it is an open-source project constantly updated. The situation of the DOT language was similar, as DOT permits the representation of graphs in an easy way, and it also permits the addition of attributes to the nodes and edges, making it perfect for this project.

The decision of using C++ as the programming language was due to the fact that it is the language where the LLVLM API is defined for, and it is the one most known by the project's

author.

The decision of using FPGAs to implement the elastic circuits, is what this approach tries to discover, if synthesizing hardware in this type of technologies is appropriate or not to improve the execution of programs with respect to CPUs and GPUs.

10.2.4. CCO3.1

To implement critical code following criteria like execution time, efficiency and security. [Enough]

As explained in the first competence, one of the main goals is reducing the time and power consumption. Therefore, we have tried to implement everything taking always into account these aspects. That is, trying to use the correct structures, as well as using the more efficient algorithms, and finding the most efficient and simpler way to implement all the functions. Also, we have ensured that there are not memory leaks or bugs. The only problem it could occur is that the program stops by triggering an assert, due to the LLVM IR code having some instructions that are not handled for some of the explained reasons.

11 Conclusions

11.1. Personal Valuation

Carrying out this project has been a good challenge as well as a good experience. In addition of learning new technical concepts, I have been able to learn and apply certain methodologies to develop projects satisfactorily.

During the degree I have been doing projects for the sake of applying and demonstrating the concepts learned in that course, obtaining a grade, and forgetting about at the moment it is delivered. They did not have any purpose nor utility. However, with this project, I have been forced to properly plan all the development, consider alternatives when something goes wrong, consider the best working methodology, and create an accurate budget that the development could suppose. And I am grateful, because during the degree these aspects are not needed to success, and this way I have been able to learn how I should carry out big and important projects from now on.

The development has gone quite well, as I have been able to fulfill all the initial objectives, as well as covering most of the planned aspects, changing some aspects for others that I considered more important, that they were not supposed to be covered, or leaving some of them uncovered

due to lack of time, or being impossible at this moment. Nonetheless, the initial plan has suffered deviations, as all the phases started before the initially planned, and all of them had a different duration than planned. Also, the development has suffered an increase of 25 hours with respect the planned duration, but most of them for the sake of better preparing the defense. Therefore, my initial thoughts about the duration of each task was a bit inaccurate. However, thanks to increasing the daily performance, and the contingency and unexpected events costs, neither the end date nor the budget have been endangered.

During all the project's life, I have learned different concepts related to the approach the complete project tries to accomplish. In addition, I have researched and learned about the different existing alternatives trying to solve the same problem or using the same concepts, all serving as a reference to this project. It is a pity that developing the complete solution was impossible, if we take into account that most of what it has been left uncovered is really interesting and completely new for me. Specially, if we consider that all the new concepts learned about the approach, have been used to understand what is behind this approach, and understand the reasons behind the work done, rather than directly applying these concepts. In the end, most of these concepts could not be very useful.

Finally, I want to express my gratitude to the director, for permitting me to carry out this project, as he was searching a master's degree student to do it. Specially if I have not been able to fully develop the solution, and only perform the first steps. Also say that he has been a great advisor, and I hope that the work I have done can help him somehow, and not be completely useless.

11.2. Future Work

Developing the complete project in the time given was quite impossible. The GEP course took an entire month, and performing other tasks at the same time was complicated. Therefore, we ended with less than three months of complete development, considering the time needed to write the documentation and prepare the defense.

As we have explained, there are some instructions in the LLVM IR that are not currently managed, and this was not our initial intention. Some of them were impossible to process, due to an incompatibility of the instruction behaviour with the hardware programming, like calling to predefined functions that we cannot have access to its definition, or those that manage dynamic memory, not existing in hardware. However, there are others that have not been considered due to lack of time. All of them are special terminator instructions that are some variation to others that we have processed, like a branch or an instruction call, or they are instructions to implement the LLVM exception-handling system. At least, these instructions are not essential, and they will appear in very specific situations.

Besides dealing with the uncovered LLVM IR instructions, the most important part of the

solution has not been covered yet. That is, the programming of the circuit's description into the FPGA. In addition of the price barrier it can have, programming an FPGA is some times more difficult than programming in a CPU or a GPU. Therefore, this supposes a considerable time of autonomous-learning, as well as the time to program and test it. These were the reasons why it was impossible to cover this aspect within the available time. However, this aspect not being covered was planned since the beginning, as it was totally impossible.

At least we have been able to cover some optimization of the graphs generated, not initially planned. These optimization simplify the graphs a lot, deleting useless elastic components. For this reason, we considered doing this instead of implementing the remaining LLVM IR instructions.

Hence, the future work consists in finding someone to finish the solution, dealing with the uncovered instructions that can be translated into hardware components, as well as implementing the elastic circuits, described by the data flow graphs, into the logic blocks of the FPGA. Then, some study should be done to decide if the results obtained in time and power consumption can be good enough to counter the drawbacks an FPGA has (e.g. programming difficulty, high cost and less application flexibility), or discard this approach. If the results were positive, and this approach promising, further research could be done to improve this approach, and use it to solve the problem in the future.

A Usage Description

In this appendix we are going to explain how the software developed in this project is executed, as well as the requisites it needs to be executed.

As we have explained, we have developed everything using the LLVM tools. In addition of the official tools, some front ends might be needed to compile different high-level languages, as LLVM only offers *clang* to compile C, C++ and Objective-C. Also, we want to mention that LLVM has multiple versions, and the latest stable is the 8. We have tested it with versions 7, 8 and 9, but in version 8 they have added some instructions in the LLVM IR. Therefore, we recommend using version 8 to execute the software.

We also need *cmake* [24] to generate *makefiles* for the different passes, and easily compile them and build the shared libraries that will execute the passes with *opt*. We have used [20], an easy tutorial to build projects with *cmake*.

Also, we want to explain how the project is structured. There are four folders, containing the different passes we have created, as well as the C++ classes to represent the data flow graphs. Then, inside each of the folder containing passes, we have structured following the guides given in [15]. We have a *cmake* file and a sub-folder containing the code of the pass, and another

cmake file. In the end, we only need to generate the libraries of the different passes, and execute them.

Below we present the list of steps that must be performed to execute the software. We explain it using Unix commands, as we have developed using an Ubuntu 18.04. However, all the needed tools are also available in Windows, so it should be possible to execute it there.

1. In the parent folder of each pass, we have to create a folder called *build*, enter inside that folder and execute the following command:

   ```
   cmake ../
   ```

   This command will generate the *makefile* in the same *build* directory, that will compile that pass and generate the mentioned shared library. Once we have the *makefile* generated, we only need to execute it using

   ```
   make
   ```

   We have to repeat this step for all the three passes we have created.

2. Once we have the shared libraries, we have to generate the LLVM IR. It will depend on the tool used, but for example in *clang* we have to execute the following command:

   ```
   clang -S -emit-llvm -Xclang -disable-O0-optnone file.c -o file.ll
   ```

   The three first options of the command are needed to generate the file containing the LLVM IR, with the *.ll* extension. The other two are needed to correctly execute the transformation passes, like *mem2reg*, otherwise they do not have any effect.

3. Once we have the LLVM IR and the passes built, we can execute the optimizer tool and execute the passes. Again, we have to execute them in the sequence we explained. For those that are already defined, we only need to specify them as a command option, but for those we have created we need to use *-load*, passing the complete path to the shared library object as argument. This library object is placed in the *build* directory, in a folder with the name of the pass. Also, the order of the passes in the command line will determine the order they are executed, and we have explained that we need a particular order. Therefore, to execute them we only need a command, but we can also divide it.

   ```
   opt -S -mem2reg -lowerswitch -load
   /path/to/the/library/object/for/gepPass -gepPass -instnamer -load
   /path/to/the/library/object/for/dfGraphPass -dfGraphPass file.ll -o
   fileMod.ll.
   ```

   As we can notice, we do not need to load the pass that computes the live variable analysis, as it is automatically executed before *dfGraphPass*. The problem is that we cannot do the same with the others, as the predefined ones do not have a header file containing the pass description, and we cannot include them in the list of requisites, as we have done with the other. With *gepPass* it would be possible, but as there is *instnamer* coming after, we cannot include it either.

B Examples

In this appendix we are going to show some examples of data flow graphs generated with the software we have developed. We will try to show different parts of the process, to facilitate the understanding of the final graph.

B.1. Example 1

We first present the source code of this example. We have omitted the main function, as it would only create some variables and call another function. This way we facilitate a bit the generated graph. This example has the following source code:

```
int elemAdd(int a[32]) {
    int x = 0;
    int i = 0;
    while (i < 32) {
        x = x + a[i];
        i = i + 1;
    }
    return x;
}

int test(int a[32], int b[32]) {
    int x = 0;
    int add1 = elemAdd(a);
    if (add1%2 == 0) {
        x = add1;
    }
    else {
        x = elemAdd(b);
    }
    return x;
}
```

B.1.1. LLVM IR

Next, we are going to show the generated and later transformed LLVM IR, using the tools we have already explained. We will show the code instead of the control-flow graph to see the different attributes the function has.

```
define dso_local i32 @elemAdd(i32* %a) #0 {
entry:
  br label %while.cond
while.cond: ; preds = %while.body, %entry
  %x.0 = phi i32 [ 0, %entry ], [ %add, %while.body ]
  %i.0 = phi i32 [ 0, %entry ], [ %add1, %while.body ]
  %cmp = icmp slt i32 %i.0, 32
  br i1 %cmp, label %while.body, label %while.end
while.body: ; preds = %while.cond
  %idxprom = sext i32 %i.0 to i64
  %arrayidx = getelementptr inbounds i32, i32* %a,
    i64 %idxprom
  %0 = load i32, i32* %arrayidx, align 4
  %add = add nsw i32 %x.0, %0
  %add1 = add nsw i32 %i.0, 1
  br label %while.cond
while.end: ; preds = %while.cond
  ret i32 %x.0
}
```

```
define dso_local i32 @test(i32* %a, i32* %b) #0 {
entry:
  %call = call i32 @elemAdd(i32* %a)
  %rem = srem i32 %call, 2
  %cmp = icmp eq i32 %rem, 0
  br i1 %cmp, label %if.then, label %if.else
if.then: ; preds = %entry
  br label %if.end
if.else: ; preds = %entry
  %call1 = call i32 @elemAdd(i32* %b)
  br label %if.end
if.end: ; preds = %if.else, %if.then
  %x.0 = phi i32 [ %call, %if.then ],
    [ %call1, %if.else ]
  ret i32 %x.0
}
```

Figure 25: Example 1 - LLVM IR (Source: own compilation)

B.1.2. Live Variable Analysis

Then, we are going to print the results of the live variable analysis, as it will be necessary for the generation of the data flow graph. We show the results of *elemAdd* aligned to the left, and the results of *text*) aligned to the right.

```
Block entry
    Live In
        a
    Live Out
        a
Block while.cond
    Live In
        a
    Live Out
        a
        x.0
        i.0
```

```
Block while.body
    Live In
        a
        x.0
        i.0
    Live Out
        a
        add
        add1
Block while.end
    Live In
        x.0
    Live Out
```

```
Block entry
    Live In
        a
        b
    Live Out
        b
        call
Block if.then
    Live In
        call
    Live Out
        call
Block if.else
    Live In
        b
    Live Out
        call1
Block if.end
    Live In
    Live Out
```

Figure 26: Example 1 - Live variable analysis (Source: own compilation)

B.1.3. Data Flow Graph

Finally, we present the data flow graph of that example. We will not show the description of that graph in DOT, as it is quite big. However, we will show it in another example.

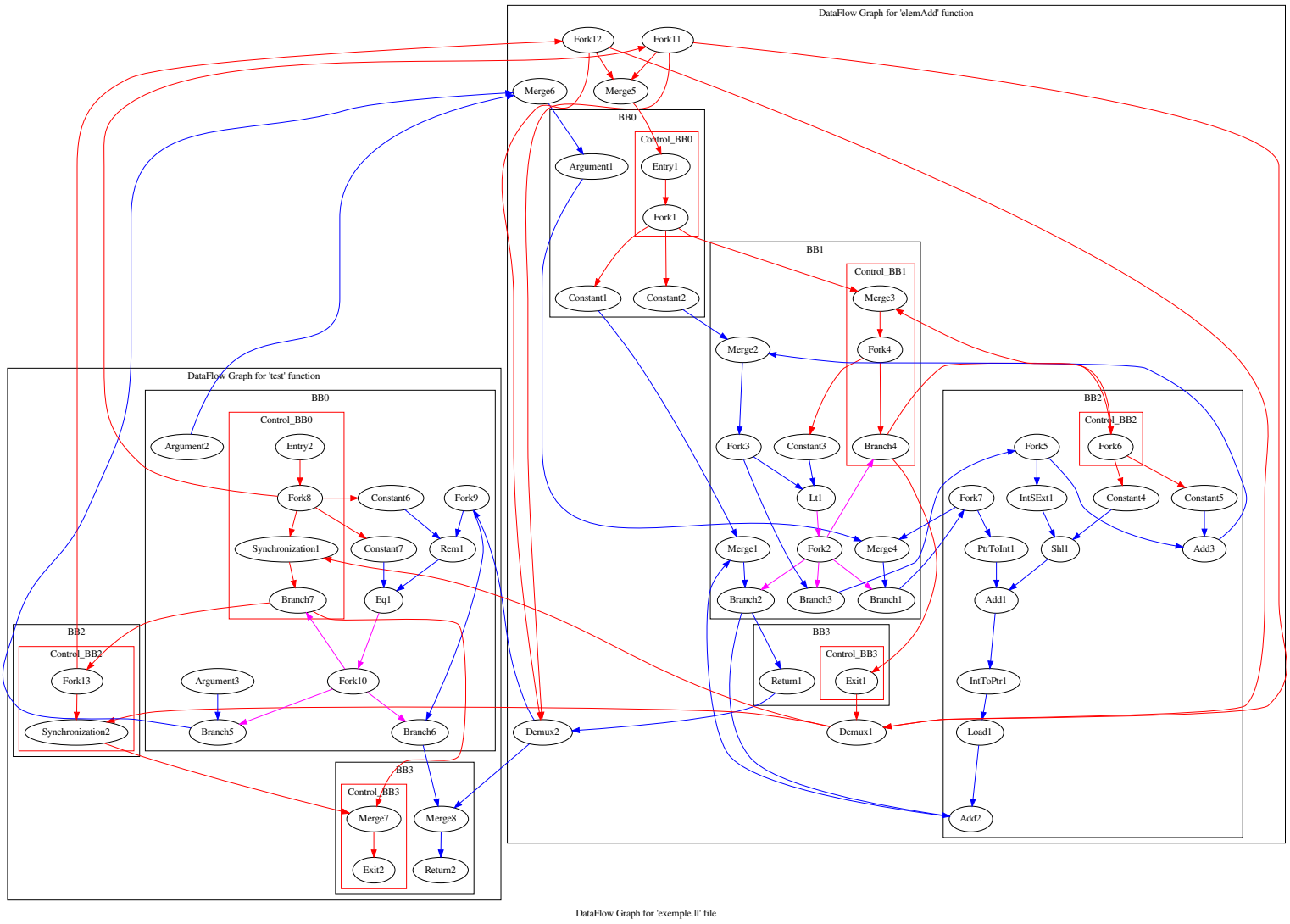

Figure 27: Example 1 - Data flow graph (Source: own compilation)

B.2. Example 2

Like we have done with the previous example, we first present the source code.

```
int gcd(int a, int b) {
    int aux = 0;
    while (b != 0) {
        aux = b;
        b = a % b;
        a = aux;
    }
    return a;
}

int main() {
    int x = gcd(52, 76);
}
```

B.2.1. LLVM IR

We will follow the same sequence, so we are going to show the generated and later transformed LLVM IR.

```
define dso_local i32 @gcd(i32 %a, i32 %b) #0 {
entry:
  br label %while.cond
while.cond: ; preds = %while.body, %entry
  %b.addr.0 = phi i32 [ %b, %entry ],
    [ %rem, %while.body ]
  %a.addr.0 = phi i32 [ %a, %entry ],
    [ %b.addr.0, %while.body ]
  %cmp = icmp ne i32 %b.addr.0, 0
  br i1 %cmp, label %while.body, label %while.end
while.body: ; preds = %while.cond
  %rem = srem i32 %a.addr.0, %b.addr.0
  br label %while.cond

while.end: ; preds = %while.cond
  ret i32 %a.addr.0
}
```

```
define dso_local i32 @main() #0 {
entry:
  %call = call i32 @gcd(i32 52, i32 76)
  ret i32 0
}
```

Figure 28: Example 2 - LLVM IR (Source: own compilation)

B.2.2. Live Variable Analysis

In this example we will only show the live variable analysis of the function that computes the great common divisor, as the main function has only one basic block, and the live variable analysis is useless in that function.

```
Block entry
    Live In
        a
        b
    Live Out
        a
        b
Block while.cond
    Live In
    Live Out
        b.addr.0
        a.addr.0
Block while.body
    Live In
        b.addr.0
        a.addr.0
    Live Out
        b.addr.0
        rem
Block while.end
    Live In
        a.addr.0
    Live Out
```

Figure 29: Example 2 - Live variable analysis (Source: own compilation)

B.2.3. Data Flow Graph

In this example we will show the resulting graph, as well as its description using the DOT language. We first show the DOT description and then the rendered graph.

```
digraph "DataFlow Graph for 'gcd.ll' file" {
    label="DataFlow Graph for 'gcd.ll' file";

    subgraph cluster_gcd {
        label="DataFlow Graph for 'gcd' function";
        subgraph cluster_BB0 {
            Argument1[type = Entry, in = "in:32", out = "out:32"];
            Argument2[type = Entry, in = "in:32", out = "out:32"];
            subgraph cluster_Control_BB0 {
                Entry1[type = Entry, in = "in:0", out = "out:0"];
                label = "Control_BB0"
                color = red
            }
            label = "BB0"
        }
        subgraph cluster_BB1 {
            Merge2[type = Merge, in = "in0:32 in1:32", out = "out:32"];
            Merge3[type = Merge, in = "in0:32 in1:32", out = "out:32"];
            Constant1[type = Constant, in = "in:0", out = "out:32", value = 0];
            Ne1[type = Operator, in = "in0:32 in1:32", out = "out:1", op = ne, latency = 0, II = 0];
            Fork1[type = Fork, in = "in:32", out = "out0:32 out1:32"];
            Branch1[type = Branch, in = "in:32 condition?:1", out = "outTrue+:32 outFalse-:32"];
            Fork2[type = Fork, in = "in:1", out = "out0:1 out1:1 out2:1"];
            Branch2[type = Branch, in = "in:32 condition?:1", out = "outTrue+:32 outFalse-:32"];
            subgraph cluster_Control_BB1 {
                Merge1[type = Merge, in = "in0:0 in1:0", out = "out:0"];
                Fork3[type = Fork, in = "in:0", out = "out0:0 out1:0"];
                Branch3[type = Branch, in = "in:0 condition?:1", out = "outTrue+:0 outFalse-:0"];
                label = "Control_BB1"
                color = red
            }
            label = "BB1"
        }
        subgraph cluster_BB2 {
            Rem1[type = Operator, in = "in0:32 in1:32", out = "out:32", op = rem, latency = 0, II = 0];
            Fork4[type = Fork, in = "in:32", out = "out0:32 out1:32"];
            label = "BB2"
        }
        subgraph cluster_BB3 {
            Return1[type = Exit, in = "in:32", out = "out:32"];
            subgraph cluster_Control_BB3 {
                Exit1[type = Exit, in = "in:0", out = "out:0"];
                label = "Control_BB3"
                color = red
            }
            label = "BB3"
        }
```

```
    }


    subgraph cluster_main {
        label="DataFlow Graph for 'main' function";
        subgraph cluster_BB0 {
            Constant2[type = Constant, in = "in:0", out = "out:32", value = 52];
            Constant3[type = Constant, in = "in:0", out = "out:32", value = 76];
            Constant4[type = Constant, in = "in:0", out = "out:32", value = 0];
            Return2[type = Exit, in = "in:32", out = "out:32"];
            subgraph cluster_Control_BB0 {
                Entry2[type = Entry, in = "in:0", out = "out:0"];
                Fork5[type = Fork, in = "in:0", out = "out0:0 out1:0 out2:0 out3:0 out4:0"];
                Synchronization1[type = Operator, in = "in0:0 in1:0", out = "out:0", op = synchronization, latency = 0, II = 0];
                Exit2[type = Exit, in = "in:0", out = "out:0"];
                label = "Control_BB0"
                color = red
            }
            label = "BB0"
        }
    }

    // gcd Channels

    // BB0
    Argument1 -> Merge3 [from = out, to = in0, color = blue];
    Argument2 -> Merge2 [from = out, to = in0, color = blue];
    // Control_BB0
    Entry1 -> Merge1 [from = out, to = in1, color = red];

    // BB1
    Merge2 -> Fork1 [from = out, to = in, color = blue];
    Merge3 -> Branch2 [from = out, to = in, color = blue];
    Constant1 -> Ne1 [from = out, to = in1, color = blue];
    Ne1 -> Fork2 [from = out, to = in, color = magenta];
    Fork1 -> Ne1 [from = out0, to = in0, color = blue];
    Fork1 -> Branch1 [from = out1, to = in, color = blue];
    Branch1 -> Fork4 [from = outTrue, to = in, color = blue];
    Fork2 -> Branch1 [from = out0, to = condition, color = magenta];
    Fork2 -> Branch2 [from = out1, to = condition, color = magenta];
    Fork2 -> Branch3 [from = out2, to = condition, color = magenta];
    Branch2 -> Rem1 [from = outTrue, to = in0, color = blue];
    Branch2 -> Return1 [from = outFalse, to = in, color = blue];
    // Control_BB1
    Merge1 -> Fork3 [from = out, to = in, color = red];
    Fork3 -> Constant1 [from = out0, to = in, color = red];
    Fork3 -> Branch3 [from = out1, to = in, color = red];
    Branch3 -> Merge1 [from = outTrue, to = in0, color = red];
    Branch3 -> Exit1 [from = outFalse, to = in, color = red];

    // BB2
    Rem1 -> Merge2 [from = out, to = in1, color = blue];
    Fork4 -> Rem1 [from = out0, to = in1, color = blue];
    Fork4 -> Merge3 [from = out1, to = in1, color = blue];

    // BB3
    // Control_BB3
    Exit1 -> Synchronization1 [from = out, to = in0, color = red];

    // main Channels

    // BB0
    Constant2 -> Argument1 [from = out, to = in, color = blue];
    Constant3 -> Argument2 [from = out, to = in, color = blue];
    Constant4 -> Return2 [from = out, to = in, color = blue];
    // Control_BB0
    Entry2 -> Fork5 [from = out, to = in, color = red];
    Fork5 -> Entry1 [from = out0, to = in, color = red];
    Fork5 -> Constant2 [from = out1, to = in, color = red];
    Fork5 -> Constant3 [from = out2, to = in, color = red];
    Fork5 -> Constant4 [from = out3, to = in, color = red];
    Fork5 -> Synchronization1 [from = out4, to = in1, color = red];
    Synchronization1 -> Exit2 [from = out, to = in, color = red];

}
```

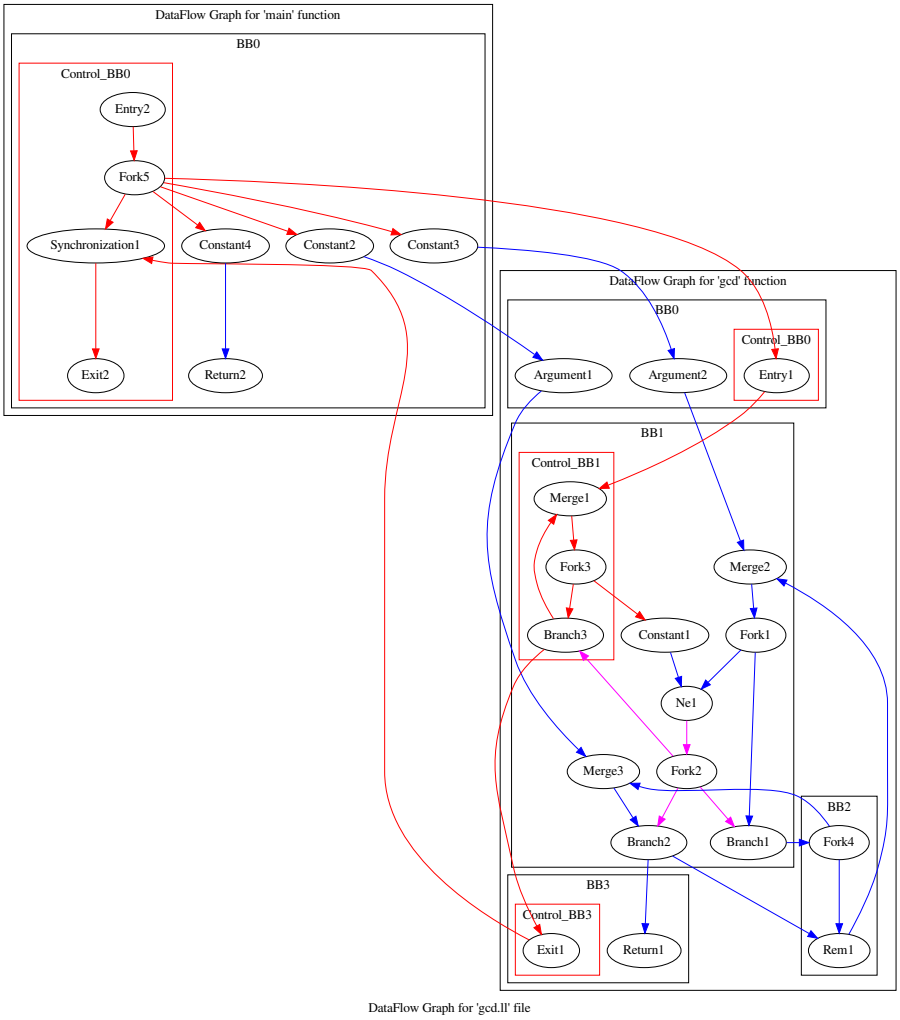

Figure 30: Example 2 - Data flow graph (Source: own compilation)

12 References

[1] Lana Josipović, Radhika Ghosal, and Paolo Ienne. *Dynamically Scheduled High-level Synthesis*. Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences.

[2] Jack B. Dennis. *First Version of a Data Flow Procedure Language*. Lecture Notes in Computer Science , vol. 19, Springer-Verlag, pp. 362-376

[3] Jack B. Dennis. *A Preliminary Architecture for a Basic Data-flow Processor*. Laboratory for Computer Science, MIT, Technical Magazine TM-61, 22 pp.

[4] Jack B. Dennis. *Data Flow Supercomputers*. Laboratory for Computer Science, MIT.

[5] Jorge E. Rodriguez. *A Graph Model for Parallel Computations*. Laboratory for Computer Science, MIT, Technical Report TR-64, 120 pp.

[6] Richard M. Karp and Raymond E. Miller. *Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing*. SIAM Journal on Applied Mathematics, vol. 14, no. 6, pp. 1390–1411.

[7] Philippe Coussy, Daniel D. Gajski, Michael Meredith and Andres Takach. *An Introduction to High-Level Synthesis*. IEEE Design & Test of Computers, vol. 26, no. 4, pp. 8-17.

[8] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown and Tomasz Czajkowsk. *LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems*. ECE Department, University of Toronto, Altera Toronto Technology Centre.

[9] Josep Carmona, Jordi Cortadella, Mike Kishinevsky and Alexander Taubin. *Elastic Circuits*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 10, pp. 1437-1455.

[10] Jordi Cortadella, Mike Kishinevsky and Bill Grundmann. *Synthesis of synchronous elastic architectures*. 2006 43rd ACM/IEEE Design Automation Conference, San Francisco, CA, 2006, pp. 657-662.

[11] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compilers Principles, Techniques, & Tools*. Addison-Wesley, Second Edition.

[12] Mohamed Ammar Ben Khadra, Yu Bai and Klaus Schneide. *High level modeling of elastic circuits in SystemC*. SpringSim 2014, Symposium on Theory of Modeling and Simulation.

[13] Mahdi Jelodari Mamaghani. *High-Level Synthesis of Elasticity: From Models to Circuits*. PhD Computer Science thesis, The University of Manchester.

[14] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea and Seth C. Goldstein. *C to Asynchronous Dataflow Circuits: An End-to-End Toolflow*. Computer Science Department, Carnegie Mellon University.

[15] LLVM. *Writing an LLVM Pass*. http://llvm.org/docs/WritingAnLLVMPass.html.

[16] LLVM. *LLVM's Analysis and Transform Passes*. `https://releases.llvm.org/7.0.0/docs/Passes.html`.

[17] LLVM. *C++ API Documentation*. `https://llvm.org/doxygen/index.html`.

[18] Graphviz. *The DOT Language*. `https://graphviz.gitlab.io/_pages/doc/info/lang.html`.

[19] Graphviz. *Node, Edge and Graph Attributes*. `https://www.graphviz.org/doc/info/attrs.html`.

[20] *Introduction to CMake by Example*. http://derekmolloy.ie/hello-world-introductions-to-cmake/.

[21] *Visul Studo Code*. https://code.visualstudio.com/.

[22] *LLVM Compiler Framework*. https://llvm.org/.

[23] *Graphviz*. https://www.graphviz.org/.

[24] *Cmake*. https://cmake.org/.

[25] *Git*. https://git-scm.com/.

[26] *Github*. https://github.com/.

[27] *Trello*. https://trello.com/.

[28] *Skype*. https://www.skype.com/.

[29] *Overleaf*. https://www.overleaf.com/.

[30] *Google Slides*. https://www.google.es/intl/es/slides/about/.

[31] *Gantter (Trial Version)*. https://www.gantter.com/.